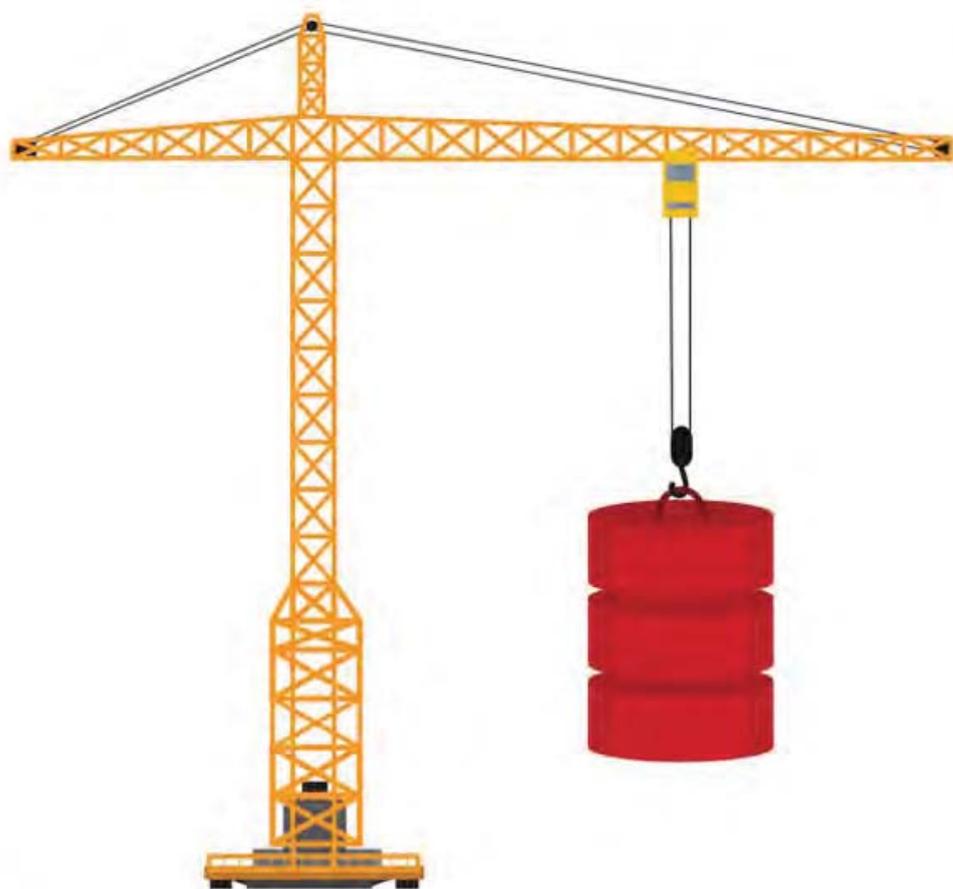
SQL

Uma abordagem para banco de dados Oracle

EDUARDO GONÇALVES

Agradecimentos

Em primeiro lugar, agradeço a Deus por este presente que é a minha vida e por mais esta conquista. Dedico este livro a minha esposa Thais e a minha filha Lisyê. Vocês são meus tesouros. Obrigado por entender as ausências, cujo motivo, foi às incontáveis horas despendidas para que este projeto fosse possível. E por último, não menos importante, agradeço aos meus pais por tudo que sempre fizeram e continuam fazendo por mim. Amo todos vocês.

Sobre o autor

Formado em Tecnologia da Informação, possui mais de 10 anos de experiência em análise e desenvolvimentos de sistema voltados á tecnologias Oracle, trabalhando em grandes empresas como Lojas Renner, Mundial S.A, Tigre S.A, Pernambucanas, Tractebel Energia, Portobello Cerâmica, Bematech, entre outros. É instrutor de cursos oficiais da Oracle – nas linguagens SQL e PL/SQL. Também atua no desenvolvimento de aplicações mobile para a plataforma iOS. Atualmente, desempenhando o papel de Líder Operacional de Projetos na Supero Tecnologia.

Prefácio

Sempre gostei de livros dos quais "conversam" com o leitor, principalmente os técnicos. Como se fosse uma troca de idéias, entre o autor e quem esta lendo. Procurei escrever desta forma, pois creio que com isto a leitura se torna mais clara e amigável, como se fosse um bate papo entre amigos, conversando sobre qualquer assunto.

Procurei colocar neste livro tudo que você vai precisar saber sobre a linguagem SQL. O livro aborda conceitos que são utilizados no dia a dia do desenvolvimento e análise de sistemas para banco de dados Oracle. Repleto de exemplos, este livro também vai ajudá-lo como fonte de referência para consulta de comandos e como utilizá-los. Tenha todos, uma ótima leitura!

Considerações

Público alvo

Este livro se destina a iniciantes e experientes na linguagem SQL. Para os iniciantes, são abordados conceitos sobre a estrutura da linguagem SQL e as características voltadas para banco de dados Oracle. Inclusive alguns conceitos do padrão ANSI, também são abordados. Para os já experientes, ele ajudará como fonte de referência e para relembrar conceitos e técnicas da linguagem.

Como está dividido o livro?

Primeiramente, são abordados conceitos sobre bancos de dados relacionais, como por exemplo, os conceitos de Relações, Tabelas, Linhas, Colunas, Registros, Integridade de Dados, bem como aspectos relacionados à Linguagem SQL, como sua definição e suas três estruturas – Linguagem de Definição dos Dados (DDL), Linguagem de Manipulação dos Dados (DML) e Linguagem de Controle dos Dados (DCL).

Depois desta introdução, são abordados e detalhados todos os aspectos da linguagem SQL no âmbito prático, onde é mostrada cada uma de suas estruturas, seus comandos e características. Sempre apresentando explicações para cada conceito ou comando e, em seguida, demonstrando aplicações através de exemplos.

Os scripts de base (tabelas e dados), fontes para a execução dos exemplos do livro, estão disponíveis no endereço https://github.com/eduardogoncalvesbr/livrosql-casadocodigo.

Contatos

Para falar com o autor, envie email para eduardogoncalves.br@gmail.com.

Sumário

Capítulo 1

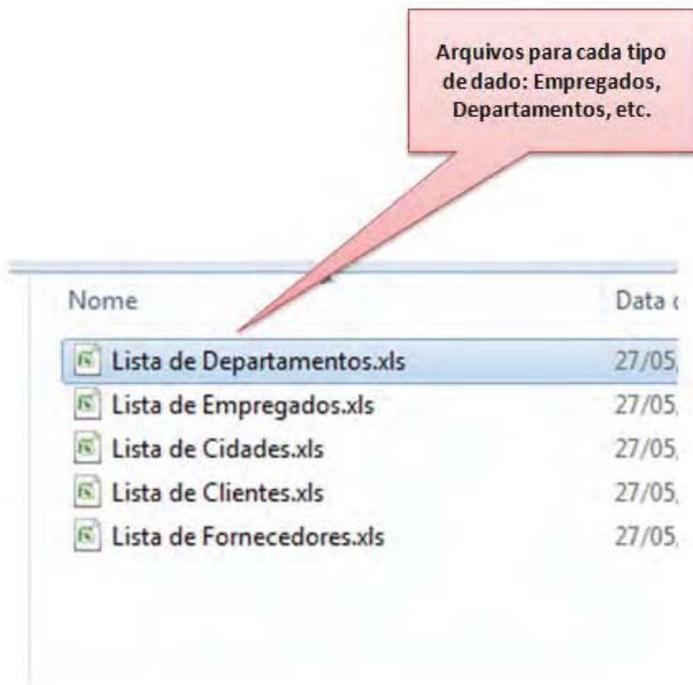
No início... Era o caos!

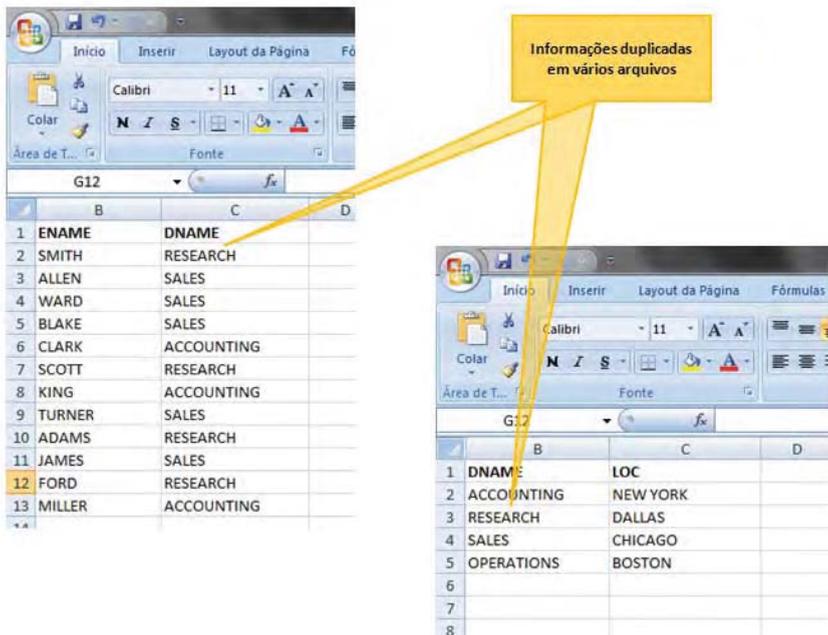
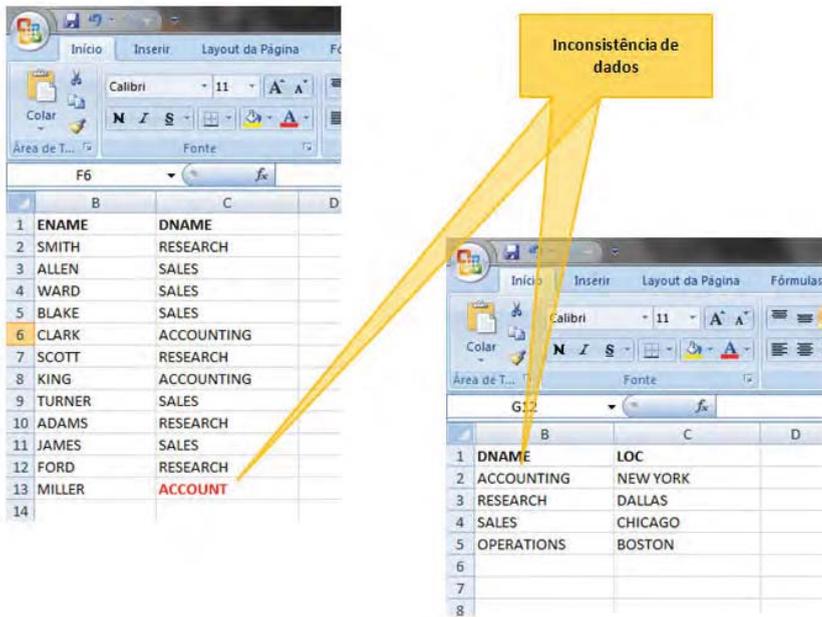
1.1 Porque ler este livro

Antes de iniciarmos com as abordagens de banco de dados e SQL vamos entender quais foram às motivações que levaram ao desenvolvimento destas tecnologias e como isso melhorou muito a forma de desenvolver sistemas, criando softwares mais robustos e confiáveis.

Houve um tempo onde não existia o conceito de banco de dados, muito menos sistemas gerenciadores para este fim. Mas então, como os dados eram armazenados? Pois bem, neste tempo os dados eram armazenados em arquivos, como por exemplo, planilhas ou arquivos de texto, que ficavam gravados nos computadores dos usuários ou em fitas e disquetes. Na época isso já era uma evolução em tanto, contudo, mesmo as informações estando armazenadas em meios digitais, havia falhas e isso ocasionava muitos problemas.

Muitos dos problemas estavam relacionados a duplicidade de dados e in-

consistências de informações. Sem falar nos problemas de falhas dos arquivos, onde os mesmos, por alguma razão acabavam corrompidos e todas as informações eram perdidas. Muitas vezes estas informações ficavam guardadas em disquetes ou fitas que se perdiam ao longo do tempo ou se degradavam pelo mau uso.

Outra dificuldade era que as informações não se relacionavam, ou seja, não existia uma ligação entre elas, e com isto, pouco poderia ser feito para utilizá-las como fonte estratégica de negócio numa empresa, por exemplo. Era muito complicado cruzar informações de diversas áreas de negócio e consolidá-las a fim de medir a produtividade de uma empresa ou usá-las em tomadas de decisão. Para que você possa entender o que estou falando, vamos

observar os casos a seguir.

Observando a figura anterior, vemos um problema clássico no uso desta abordagem, a duplicidade de dados. Nele vemos que existem duas planilhas, uma com a lista de empregados e outra com a lista de departamentos. Além das informações estarem em planilhas independentes, não possuindo assim, referencia uma com a outra, é possível constatar a duplicidade de dados, onde o nome do departamento, neste caso, representado pela coluna DNAME, aparece nas duas listas. Isto era um problema, pois a duplicidade de dados, além de resultar em maior espaço em disco, abre margem para outro problema que era a inconsistências de dados, conforme pode ser visto no próximo exemplo.

Voltando as mesmas listas anteriores, a de empregados e departamentos, podemos observar que a escrita referente ao departamento ACCOUNTING (Contabilidade) na planilha de empregados foi escrito de forma incorreta, sendo digitado como ACCOUNT. Com isto, aumentam-se as chances do surgimento de dados inconsistentes e não confiáveis, onde temos duas descrições diferentes, mas que querem dizer a mesma coisa.

Outra forma de armazenamento de dados em arquivos era o famoso "tabelão", onde as informações de diferentes tipos eram gravadas todas num único arquivo. Veja o exemplo a seguir.

1	NOME	EMAIL	TELEFONE	DATA ADMISSAO	CARGO	SALARIO	DEPARTAMENTO	ENDERECO	CIDADE	ESTADO	REGIAO	PAIS
2	Steven King	SKING	5.151.234.567	17/06/1987	AD_PRES	24000	Executive	2004 Charade Rd	Seattle	Washington	Americas	United States of
3	Neena Kochhar	NKOCHHAR	5.151.234.568	21/09/1989	AD_VP	17000	Executive	2004 Charade Rd	Seattle	Washington	Americas	United States of
4	Lex De Haan	LDEHAAN	5.151.234.569	13/01/1993	AD_VP	17000	Executive	2004 Charade Rd	Seattle	Washington	Americas	United States of
5	Alexander Hunold	AHUNOLD	5.904.234.567	03/01/1990	IT_PROG	9000	IT	2014 Jabberwocky Rd	Southlake	Texas	Americas	United States of
6	Bruce Ernst	BERNST	5.904.234.568	21/05/1991	IT_PROG	6000	IT	2014 Jabberwocky Rd	Southlake	Texas	Americas	United States of
7	David Austin	DAUSTIN										
8	Valli Pataballa	VPATABA										
9	Diana Lorentz	DLOREN										
10	Nancy Greenberg	NGREE										
11	Daniel Faviet	DFAV										
12	John Chen	JCH										
13	Ismael Sciarra	ISC										
14	Jose Manuel Urman	J										
15	Luis Popp											
16	Den Raphaely											
17	Alexander Khoo											
18	Shelli Baida											
19	Sigal Tobias	S										
20	Guy Himuro	GH										
21	Karen Colmenares	KCO										
22	Matthew Weiss	MWE										
23	Adam Fripp	AFRIPP										
24	Payam Kaufling	PKAUFL										
25	Shanta Vollman	SVOLLMA										

	A	B	C	D	E	F	G
1	NOME	EMAIL	TELEFONE	DATA ADMISSAO	CARGO	SALARIO	DEPARTAM
2	Steven King	SKING	5.151.234.567	17/06/1987	AD_PRES	24000	Executive
3	Neena Kochhar	NKOCHHAR	5.151.234.568	21/09/1989	AD_VP	17000	Executive
4	Lex De Haan	LDEHAAN	5.151.234.569	13/01/1993	AD_VP	17000	Executive
5	Alexander Hunold	AHUNOLD	5.904.234.567	03/01/1990	IT_PROG	9000	IT
6	Bruce Ernst	BERNST	5.904.234.568	21/05/1991	IT_PROG	6000	IT
7	David Austin	DAUSTIN	5.904.234.569	25/06/1997	IT_PROG	4800	IT
8	Valli Pataballa	VPATABAL	5.904.234.560	05/02/1998	IT_PROG	4800	IT
9	Diana Lorentz	DLORENTZ	5.904.235.567	07/02/1999	IT_PROG	4200	IT
10	Nancy Greenberg	NGREENBE	5.151.244.569	17/08/1994	FI_MGR	12000	Finance
11	Daniel Faviet	DFAVIET	5.151.244.169	16/08/1994	FI_ACCOUNT	9000	Finance
12	John Chen	JCHEN	5.151.244.269	28/09/1997	FI_ACCOUNT	8200	Finance

Note que todos os dados referentes aos empregados, como, nome, departamento, cargo, etc. estão todos em um mesmo arquivo. Com isto, vemos muitas informações sendo duplicadas em um arquivo inchado, onde os riscos de inconsistência são enormes, pois a mesma informação que aparece em vários lugares, por exemplo, podem conter descrições diferentes por conta de erros de digitação. Sem falar, quando uma informação muda, como, por exemplo, um nome de departamento ou de um cargo, é necessário verificar todas as linhas onde esta informação aparece para alterá-las de forma igual. Além do retrabalho, pode ocorrer, por esquecimento de quem está atualizando, que linhas que não sejam alteradas com a nova informação, gerem dados conflitantes e sem nexo.

Creio que você pode ter uma idéia de como as coisas eram antes de existirem sistemas de gerenciamento de dados, cujo objetivo era justamente possibilitar um ambiente consistente e organizado, o que não acontecia no passado. Foi assim, para que estes e outros problemas fossem resolvidos, que surgiram os bancos de dados e seus gerenciadores. Com uma implementação baseada em técnicas de modelagem de dados e com uma linguagem robusta de acesso, foi possível criar um ambiente seguro, confiável e de alto desempenho e processamento para a manipulação e armazenagem dos dados.

Além de manter todos os dados num único lugar, de forma centralizada, através da modelagem de dados foi possível levar o conceito de administra-

ção de dados para um nível mais alto, diria até um existencial, muito melhor e muito mais organizado. Através das regras de normalização de dados, a chamadas Formas Normais, é possível modelar toda estrutura de dados de um sistema, visando a organização, o desempenho e principalmente a flexibilidade e a clareza. Quando modelamos um sistema, não apenas criamos a base de dados a satisfazer apenas o intuito de armazenamento, mas sim, avaliamos qual a melhor de fazer isto. Para exemplificar, veja como ficar o exemplo anterior, utilizando os dados de Empregados e Departamentos.

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1	7369	SMITH	CLERK	7902	17/12/1980	800,00		20
2	7499	ALLEN	SALESMAN	7698	20/02/1981	1600,00	300,00	30
3	7521	WARD	SALESMAN	7698	22/02/1981	1250,00	500,00	30
4	7566	JONES	MANAGER	7839	02/04/1981	2975,00		20
5	7654	MARTIN	SALESMAN	7698	28/09/1981	1250,00	1400,00	30
6	7698	BLAKE	MANAGER	7839	01/05/1981	2850,00		30
7	7782	CLARK	MANAGER	7839	09/06/1981	2450,00		10
8	7788	SCOTT	ANALYST	7566	09/12/1982	3000,00		20
9	7839	KING	PRESIDENT		17/11/1981	5000,00		10
10	7844	TURNER	SALESMAN	7698	08/09/1981	1500,00	0,00	30
11	7876	ADAMS	CLERK	7788	12/01/1983	1100,00		20
12	7900	JAMES	CLERK	7698	03/12/1981	950,00		30
13	7902	FORD	ANALYST	7566	03/12/1981	3000,00		20
14	7934	MILLER	CLERK	7782	23/01/1982	1300,00		10

	DEPTNO	DNAME	LOC
1	10	ACCOUNTING	NEW YORK
2	20	RESEARCH	DALLAS
3	30	SALES	CHICAGO
4	40	OPERATIONS	BOSTON

Não vamos aqui, nos aprofundar nos conceitos de modelagem de dados, entrando nos meandros de cada Forma Normal, por exemplo, nem nas demais técnicas, mas seguindo este exemplo simples, é possível demonstrar como podemos estruturar estas duas fontes de dados de forma clara e organizada. Note que nesta ultima figura tempos o que podemos chamar de duas tabelas, uma de dados de empregados, onde temos informações sobre

nomes (ENAME), cargos (JOB), salários (SAL) e outra de dados de departamento, onde temos informações com nomes (DNAME) e localizações (LOC). Veja que embora sejam duas fontes de dados distintas existe uma ligação entre elas. Esta ligação está sendo feita através de uma das colunas, a coluna DEPTNO. Veja que esta coluna existe nas duas tabelas. Este que parece ser um pequeno detalhe faz com que nossas fontes sejam consistentes e estejam dentro das regras de modelagem. Observe como ligamos vários empregados a departamento, sem repetir as informações, utilizando um código. Com isto, se necessitarmos alterar alguma informação relacionada a um determinado departamento, como, por exemplo, o nome ou sua localização, não precisamos alterar para cada empregado, alteramos apenas o departamento em questão, na tabela de departamentos. Desta forma, conseguimos manter as informações de forma consistente, sem duplicidades, e desta forma também dispostos de forma estruturada.

É obvio que este foi um exemplo simples, como já mencionado, contudo, serve para mostrar que com o advento dos bancos de dados estas e outras técnicas podem ser utilizadas na criação de sistemas capazes de atender as demandas mais complexas ou que exigem maior processamento e alta performance. Além do mais, o banco de dados permite a possibilidade de criarmos um ambiente único onde todas as fontes de dados coexistam nele. Desta forma, deixamos de ter arquivos distintos, em localizações diferentes e que não possuem uma ligação lógica, nem mesmo física.

Agora que você já entendeu como era no início, está na hora de entender o agora, como funcionando os bancos de dados e seus gerenciadores. Entender seus conceitos e como trabalham. Além disso, aprender como é feita a comunicação com estes bancos e com os dados armazenados neles.

Capítulo 2

Banco de dados

Para se obter o armazenamento adequado da informação, é preciso que sejam escolhidos métodos que satisfaçam os requisitos de segurança e consistência. O banco de dados, principalmente o baseado no modelo relacional, é muito utilizado para este fim. Para tanto, deve-se entender suas características, seus conceitos e fundamentos. A utilização dos métodos de armazenamento através do uso de tabelas (linhas e colunas) garante seu armazenamento, bem como as regras definidas para o modelo relacional. A Integridade referencial precisa ser estudada e analisada. A álgebra relacional, que se trata de uma das técnicas para recuperação de dados pelos motores dos bancos de dados, deve ser entendida e operada de forma eficaz para alcançar o objetivo esperado. E por último, mas não menos importante, deve-se conhecer a linguagem padrão de acesso a dados, a SQL, para o modelo relacional, a qual reúne as abordagens das linguagens de definição, manipulação e controle dos dados.

2.1 Introdução ao banco de dados relacional

Banco de dados relacional pode ser visto como uma coleção de dados. Esses dados estão disponibilizados de forma organizada e integrados, armazenados em forma de tabelas interligadas por chaves primárias e estrangeiras, constituindo uma representação de dados natural, podendo ser modificada sem restrições. Além disso, o banco pode ser utilizado por todas as aplicações relevantes sem duplicação de dados e sem a necessidade de serem definidos em programas, mas podendo ser, como composto de tabelas ou relações.

A abordagem relacional dos dados é baseada na observação de que arquivos podem ser considerados como relações matemáticas. Consequentemente, a teoria elementar de relações pode ser usada para lidar com vários problemas práticos que surgem com os dados desses arquivos. Com base em Edgar F. Codd, o principal expositor do modelo de dados relacional, pode-se dizer que um dos recursos poderosos do modelo relacional é a capacidade de permitir a manipulação de relações orientadas a conjuntos. Esta característica possibilitou o desenvolvimento de poderosas linguagens não procedurais com base na teoria dos conjuntos (álgebra relacional) ou em lógica (cálculo relacional). Em consequência disto, tabelas são tratadas como relações e as linhas dessas tabelas são usualmente conhecidas como tuplas, que conforme a maior parte da literatura pode ser definida como linha ou registro.

Um banco de dados relacional tem como objetivo a implementação do modelo de dados relacional, incorporando todas as características básicas como representação de entidades, atributos e relacionamentos. As entidades, também chamadas de tabelas, são constituídas por um conjunto de linhas não ordenadas, as tuplas. Cada linha que faz parte da tabela é constituída por um ou vários campos que são chamados de atributos da entidade. Vamos pegar como exemplo a tabela chamada EMP, do banco de dados que será usado em nosso treinamento:

Para que haja o relacionamento entre as entidades, mais precisamente entre as linhas de uma tabela, é preciso estabelecer o conceito de chaves. Os tipos de chaves geralmente encontradas num banco de dados relacional são três: chave primária (PK *Primary Key*), chave estrangeira (FK *Foreign Key*) e chave alternativa.

2.2 Chave primária (índice primário)

Trata-se de um identificador único para tabela. Quando uma coluna ou combinações de coluna é instituída como chave primária, nenhum par de linhas da tabela pode conter o mesmo valor naquela coluna ou combinação de colunas. Uma chave primária deve seguir o conceito da minimalidade, isto é, que todas as suas colunas sejam efetivamente necessárias para garantir o requi-

sito de unicidade de valores desta chave. Portanto, quando se consegue, com um atributo, garantir a unicidade de um registro, um segundo atributo compondo a chave é desnecessário. Quando se consegue com dois, um terceiro é desnecessário e assim por diante. Voltemos à tabela EMP:

EMPNO	ENAME	JOB	MGR
7369	SMITH	CLERK	7902
7499	ALLEN	SALESMAN	7698
7521	WARD	SALESMAN	7698
7566	JONES	MANAGER	7839
7654	MARTIN	SALESMAN	7698
7698	BLAKE	MANAGER	7839
7782	CLARK	MANAGER	7839
7788	SCOTT	ANALYST	7566
7839	KING	PRESIDENT	
7844	TURNER	SALESMAN	7698
7876	ADAMS	CLERK	7788
7900	JAMES	CLERK	7698
7902	FORD	ANALYST	7566
7934	MILLER	CLERK	7782

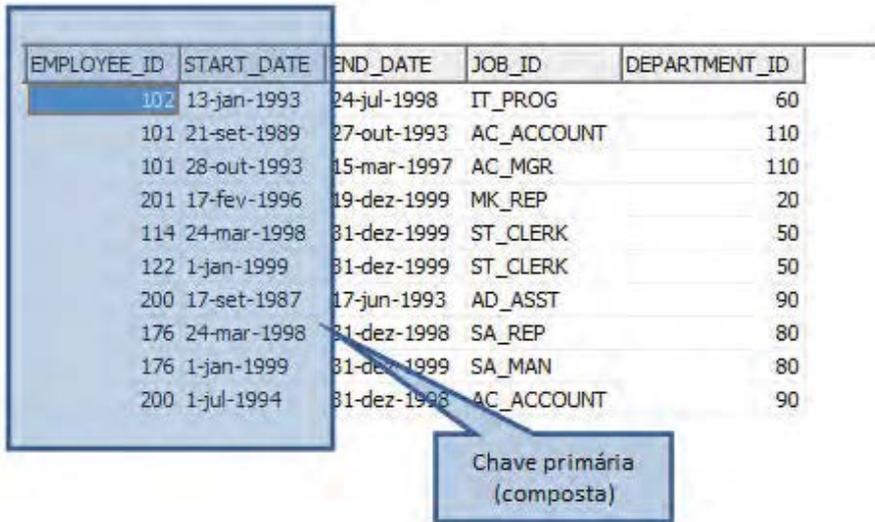
Fig. 2.2: No caso da tabela EMP, existe apenas uma coluna definida como chave primária (PK Simples). Portanto, a garantia da unicidade dos dados foi conseguida com apenas uma coluna da tabela, a coluna EMPNO.

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
102	13-jan-1993	24-jul-1998	IT_PROG	60
101	21-set-1989	27-out-1993	AC_ACCOUNT	110
101	28-out-1993	15-mar-1997	AC_MGR	110
201	17-fev-1996	19-dez-1999	MK_REP	20
114	24-mar-1998	31-dez-1999	ST_CLERK	50
122	1-jan-1999	31-dez-1999	ST_CLERK	50
200	17-set-1987	17-jun-1993	AD_ASST	90
176	24-mar-1998	31-dez-1998	SA_REP	80
176	1-jan-1999	31-dez-1999	SA_MAN	80
200	1-jul-1994	31-dez-1998	AC_ACCOUNT	90

Fig. 2.3: Neste caso foram necessárias mais de uma coluna (PK Composta) para garantir a unicidade dos dados, as colunas EMPLOYEE_ID e START_DATE.

2.3 Chave estrangeira

Esta chave é determinada por uma coluna ou um conjunto de colunas possuidoras de um conjunto de valores válidos que estão presentes em outra tabela. Para enfatizar este conceito, pode-se dizer que uma chave estrangeira é uma coluna ou uma combinação de colunas, cujos valores aparecem necessariamente na chave primária de outra tabela. A chave estrangeira pode ser vista como sendo o mecanismo de implementação do relacionamento entre tabelas de um banco de dados relacional.

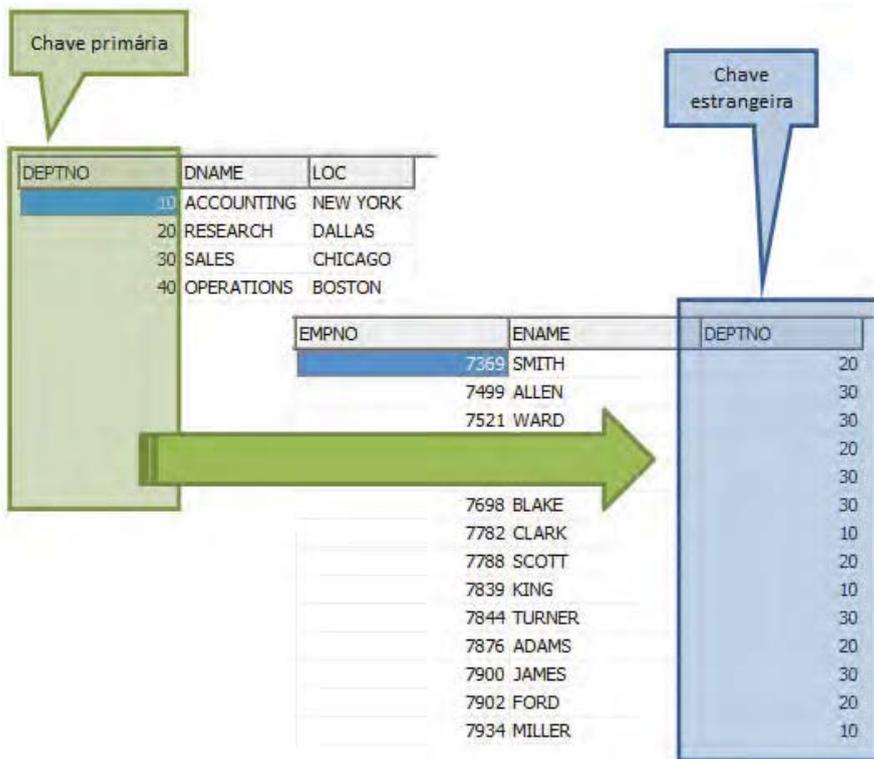



Fig. 2.4: As tabelas DEPT e EMP possuem suas colunas definidas como chave primária (EMPNO tabela EMP e DEPTNO tabela DEPT) e estão interligadas através da coluna DEPTNO, que existe tanto na tabela DEPT quanto na tabela EMP. Esta ligação garante que um empregado só poderá ser atribuído a um departamento existente na tabela de departamentos (DEPT).

Relacionamentos

As tabelas "conversam" entre si através de relacionamentos. Para que isso seja possível, é preciso que haja informações (colunas) que sejam comuns entre as tabelas. Um exemplo típico é a existência da coluna DEPTNO na tabela de empregados. A ligação ou relacionamento entre as duas tabelas é realizado através deste campo. Na figura anterior, existe um relacionamento entre as tabelas EMP e DEPT.

2.4 Chave alternativa

Vimos que existem princípios para a criação de chaves primárias. O princípio da unicidade e o da minimalidade mostram que, embora se tenha várias colunas que podem fazer parte da chave primária (também chamadas de chaves candidatas), na maioria das vezes não se faz necessário, nem adequado, o uso delas para garantir a integridade. No momento da escolha de qual coluna deve ser usada, pode-se tomar como critério a escolha da coluna que se deseja usar nas chaves estrangeiras que referenciam a tabela em questão. Essas chaves candidatas podem ser utilizadas como índices secundários, úteis para acelerar operações de busca, embora nem sempre sejam eficazes. As chaves alternativas podem ser definidas como sendo um identificador único, inclusive sendo implementado como tal em uma base de dados.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL
100	Steven	King	SKING
101	Neena	Kochhar	NKOCHHAR
102	Lex	De Haan	LDEHAAN
103	Alexander	Hunold	AHUNOLD
104	Bruce	Ernst	BERNST
105	David	Austin	DAUSTIN
106	Valli	Pataballa	VPATABAL
107	Diana	Lorentz	DLORENTZ
108	Nancy	Greenberg	NGREENBE
109	Daniel	Faviet	DFAVIET
110	John	Chen	JCHEN
111	Ismael	Sciarra	ISCIARRA
112	Jose Manuel	Urman	JMURMAN
113	Luis	Popp	LPOPP

Fig. 2.5: A tabela EMPLOYEES tem como chave primária a coluna EMPLOYEE_ID e como chave candidata a coluna EMAIL. A coluna EMAIL só conterá valores únicos, podendo ser utilizada como parâmetro em pesquisas realizadas nesta tabela.

Com relação ao uso de chaves estrangeiras, algumas regras devem ser observadas. A existência de uma chave estrangeira impõe restrições que devem ser garantidas ao executar diversas operações de alteração do banco de dados. Veja algumas a seguir:

- **Quando da inclusão de uma linha na tabela que contém a chave estrangeira:** deve ser garantido que o valor da chave estrangeira apareça na coluna da chave primária referenciada.
- **Quando a alteração do valor da chave estrangeira:** deve ser garantido que o novo valor de uma chave estrangeira apareça na coluna da chave primária referenciada.
- **Quando da exclusão de uma linha da tabela que contém a chave primária referenciada pela chave estrangeira:** deve ser garantido que na coluna chave estrangeira não apareça o valor da chave primária que está sendo excluída.
- **Quando da alteração do valor da chave primária referenciada pela chave estrangeira:** deve ser garantido que na coluna chave estrangeira não apareça o antigo valor da chave primária que está sendo alterada.

Outro aspecto que deve ser observado é a possibilidade de haver uma chave estrangeira que esteja referenciando a chave primária da própria tabela. Quando isso acontece, a solução se dá por meio do autorrelacionamento, isto é, a associação acontece na mesma tabela. A tabela-mãe e a tabela-filha são a mesma tabela. Vale ressaltar que uma chave estrangeira, sendo ela autorrelacionada ou não, pode conter valores nulos.

Chave estrangeira referencia a chave primária na mesma tabela

EMPNO	ENAME	JOB	MGR	HIREDATE
7369	SMITH	CLERK	7902	17-dez-1980
7499	ALLEN	SALESMAN	7698	20-fev-1981
7521	WARD	SALESMAN	7698	22-fev-1981
7566	JONES	MANAGER	7839	2-abr-1981
7654	MARTIN	SALESMAN	7698	28-set-1981
7698	BLAKE	MANAGER	7839	1-mai-1981
7782	CLARK	MANAGER	7839	9-jun-1981
7788	SCOTT	ANALYST	7566	9-dez-1982
7839	KING	PRESIDENT		17-nov-1981
7844	TURNER	SALESMAN	7698	8-set-1981
7876	ADAMS	CLERK	7788	12-jan-1983
7900	JAMES	CLERK	7698	3-dez-1981
7902	FORD	ANALYST	7566	3-dez-1981
7934	MILLER	CLERK	7782	23-jan-1982

Fig. 2.6: Todos os empregados da tabela EMP possuem um identificador (EMPNO). Entretanto, há outros que, além deste código, também possuem outro identificador que define quem é seu gerente. Neste exemplo, podemos perceber que o empregado ALLEN tem como gerente (MGR) o também empregado BLAKE. Neste nosso modelo, o autorrelacionamento acontece quando um empregado tem como gerente um empregado também cadastrado na mesma tabela, recebendo assim um código que existe na coluna EMPNO.

Quando se fala no uso de chaves, surge outro ponto importante, que é sobre o conceito de **regras de integridade**. O modelo relacional de banco de dados prevê duas regras gerais de integridade. São conhecidas como integridade de entidade e integridade referencial. De modo genérico, essas duas regras são aplicáveis a qualquer banco de dados que venha a implementar o modelo relacional.

Se os dados de um banco de dados estão íntegros, isso significa dizer que eles refletem corretamente a realidade representada pelo banco de dados e que são consistentes entre si. A integridade visa proteger os dados de um banco provendo sua segurança e integridade para que as informações sejam armazenadas de forma adequada. Com relação aos dois tipos de regras de integridade, pode-se dizer que:

2.5 Integridade de entidade

Um atributo, que faz parte de uma chave primária em uma relação, não poderá conter valores nulos, ou seja, ausência de informação. Outra característica importante da integridade de entidade diz respeito aos dados que são armazenados, pois os valores contidos nos campos de chave primária devem ser únicos, não havendo duplicidade de dados.

2.6 Integridade referencial

Esta integridade se trata de uma restrição que fixa que os valores que estão contidos em um campo de chave estrangeira devem, por sua vez, aparecer nos campos de chave primária da tabela referenciada. Sua principal função é garantir que as referências (ligações) entre as relações, relacionamentos entre tabelas, estejam compatíveis.

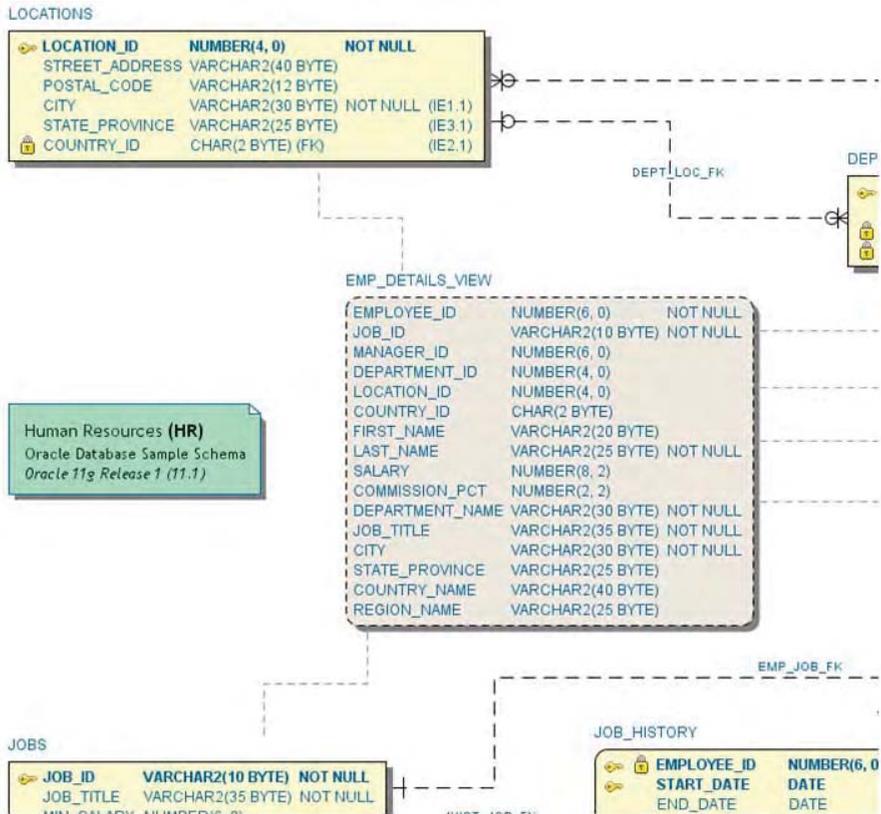



Fig. 2.7: A figura mostra com clareza as conexões existentes entre as tabelas LOCATIONS, JOBS, JOB_HISTORY, entre outras, através das chaves estrangeiras. Estas ligações garantem que os dados armazenados estejam consistentes e confiáveis. Trata-se de um exemplo, onde são mostradas as chaves primárias e estrangeiras existindo entre as tabelas.

Para que as regras de integridade sejam aplicadas de forma eficaz, é necessário que haja um sistema que as gerencie. Neste momento, entra em jogo o SGBD (Sistema de Gerenciamento de banco de dados). Um dos objetivos primordiais de um SGBD é a integridade de dados. Todas as regras de restrição existentes aplicáveis a um banco de dados são controladas por este sistema, ou seja, ele garante que as elas sejam aplicadas de forma a manter consistentes

os dados. De forma sucinta, pode-se dizer que o sistema de gerenciamento do banco de dados (também podendo ser chamado de DBMS) é o software que controla e manipula todos os acessos ao banco de dados. Alguns benefícios dos SGDBs são os seguintes:

- Simplicidade e uniformidade (o modelo relacional é compacto);
- Independência total dos dados (benefício mais importante);
- Interfaces de alto nível para usuários finais;
- Visões múltiplas de dados;
- Melhoria do diálogo entre o CPD e o usuário;
- Melhoria na segurança dos dados;
- Redução significativa do atravancamento de aplicações e do tempo;
- Gastos na manutenção;
- Alívio da carga de trabalho do CPD;
- Possibilidade de crescimento futuro e inclusão de novos dados devido à flexibilidade do sistema e independência de dados físicos.

Os SGBDs seguem também princípios que os caracterizam como um sistema de gerenciamento de bancos de dados relacionais. Em 1970, E. F. Codd, matemático da IBM já mencionado anteriormente, publicou um artigo intitulado como "Um modelo relacional de dados para grandes bancos de dados compartilhados", contendo os princípios, ou regras, que definem um sistema de gerenciamento de bancos de dados como sendo um sistema relacional. São as que seguem:

- **Regra 1 :** todas as informações em um banco de dados relacional são representadas de forma explícita no nível lógico e exatamente em apenas uma forma, ou seja, por valores em tabelas.

- **Regra 2 :** cada um e qualquer valor individual (atômico) em um banco de dados relacional possui a garantia de ser logicamente acessado pela combinação do nome da tabela, no valor da chave primária e do nome da coluna.

- **Regra 3 :** valores nulos devem ser suportados de forma sistemática e independente do tipo de dados usado para representar informações inexistentes e informações inaplicáveis.

- **Regra 4 :** a descrição do banco de dados é representada no nível lógico da mesma forma que os dados ordinários, permitindo aos usuários autorizados que utilizem a mesma linguagem relacional aplicada aos dados regulares.

- **Regra 5 :** um sistema relacional pode suportar várias linguagens e várias formas de recuperação de informações, entretanto, deve haver pelo menos uma linguagem, como uma sintaxe bem definida e expressa por conjuntos de caracteres, que suporte de forma compreensiva todos os seguintes itens: definição de dados, definição de "views", manipulação de dados (interativa e embutida em programas), restrições de integridade, autorizações e limites de transações (begin, commit e rollback).

- **Regra 6 :** todas as "views", que são teoricamente atualizáveis, devem ser também atualizáveis pelo sistema.

- **Regra 7 :** a capacidade de manipular um conjunto de dados (relação) através de um simples comando deve-se entender as operações de inclusão, alteração ou exclusão de dados.

- **Regra 8 :** programas de aplicação permanecem logicamente inalterados quando ocorrem mudanças no método de acesso ou forma de armazenamento físico.

- **Regra 9 :** mudanças nas relações e nas "views" devem provocar o mínimo impacto possível nas aplicações.

- **Regra 10 :** as aplicações não podem ser afetadas quando ocorrem mudanças nas regras de restrições de integridade.

- **Regra 11 :** as aplicações não podem ser logicamente afetadas quando ocorrem mudanças geográficas nos dados.
- **Regra 12 :** se um sistema possui uma linguagem de baixo nível, essa linguagem não pode ser usada para subverter as regras de integridade e restrições definidas no nível mais alto.

Além destas regras, Codd também descreveu uma álgebra relacional fundamentando este sistema de gerenciamento de banco de dados relacional.

Capítulo 3

Introdução à linguagem SQL

A SQL (*Structured Query Language*) é uma linguagem para interface com bancos de dados relacionais, isto é, todos os usuários e programas que desejarem realizar alguma tarefa no banco de dados devem fornecer comandos escritos nesta linguagem. Existem inúmeras versões de SQL, mas a versão original foi desenvolvida no laboratório de pesquisa da IBM San Jose (atualmente Centro de Pesquisa Almaden). No início, por volta de 1970, foi chamada de SEQUEL e depois o nome foi mudado para SQL. A linguagem SQL foi desenvolvida para ser usada segundo o modelo de Codd e permanece até hoje como a linguagem padrão para banco de dados relacionais. Em 1986, foi estabelecido um padrão pelo ANSI (*American National Standards Institute*) o chamado SQL1. Com o tempo, foram aparecendo outras necessidades e foram atribuídas outras funcionalidades à linguagem, sendo criado o padrão SQL2. Atualmente, a linguagem SQL está no padrão SQL3, cujas principais inovações são a criação de funções, regras, procedimentos, armazenamento

de imagens e sons em tabelas, disparadores de eventos (gatilhos), armazenamento de códigos em Java, entre outras.

A SQL inclui, em suas fundamentações, as operações da álgebra relacional. Por exemplo: em SQL, tem-se o comando `SELECT` que faz o papel do operador de Projeção em álgebra relacional. Já os comandos `FROM` e `WHERE` da linguagem SQL funcionam como referência aos operadores de Produto cartesiano e de Seleção, respectivamente. O nascimento da linguagem SQL deveu-se à necessidade de executar as operações relacionais, incluindo Diferença, Divisão, Intersecção, Junção, Produto cartesiano, Projeção e União. Esta linguagem é flexível e economiza tempo. Praticamente todos os fornecedores de SGBDR (Sistema Gerenciador de Banco de Dados Relacional) oferecem ferramentas de 4ª Geração (4GL) que estão aptas a trabalhar com esta linguagem, com a qual podem realizar repetidos testes e posteriormente incluí-los em códigos de aplicativos.

Com relação à abrangência da linguagem SQL, pode-se dizer que ela possui inúmeras funcionalidades, dentre as quais está a gama de comandos que possui – Comandos de manipulação, Comandos de definição e Controle.

Linguagem de Definição dos Dados (Data Definition Language DDL)

Esta parte da SQL descreve como as tabelas e outros objetos podem ser definidos. Os comandos da linguagem de definição de dados (DDL) são responsáveis por definir tabelas e outros objetos pertencentes ao banco dados. Dentre os principais comandos da DDL estão: `CREATE TABLE`, `CREATE INDEX`, `ALTER TABLE`, `DROP TABLE`, `DROP VIEW` E `DROP INDEX`.

Linguagem de Manipulação dos Dados (Data Manipulation Language – DML)

Esta parte da SQL possui comandos para que os usuários façam acesso e armazenamento dos seus dados. Fazem parte da DML os seguintes comandos: `INSERT`, `DELETE`, `UPDATE`, `SELECT` E `LOCK`.

Linguagem de Controle dos Dados (Data Control Language DCL)

Para finalizar, temos a DCL, que é subdividida em três grupos de co-

mandos: comandos para controle de transação, sessão e sistema. Para os comandos de controle, podem-se citar os mais usados, que são: `COMMIT`, `ROLLBACK`, `GRANT` e `REVOKE`.

Quem pode utilizar a Linguagem SQL e quais suas vantagens?

A linguagem SQL pode ser usada por aqueles que necessitam se comunicar com bancos de dados relacionais. Trata-se de uma linguagem flexível, simples, mas ao mesmo tempo completa e de simples interpretação. Possui grandes recursos para as mais variadas operações envolvendo manipulação de dados. É de fácil escrita e leitura, pois permite a compreensão das instruções mesmo para aqueles que possuem pouco conhecimento nesta linguagem. Contudo, embora pareça simplória, seu poderio garante sua utilização pelos maiores bancos de dados do mercado.

3.1 Oracle Corporation e Banco de dados Oracle

Em 1977, a empresa Software Development Laboratories (SDL) foi formada por Larry Ellison, Bob Miner, Ed Oates e Bruce Scott. Larry e Bob vieram da Ampex, onde estavam trabalhando em um projeto da CIA apelidado de "Oracle". A CIA foi seu primeiro cliente, embora o produto ainda não tivesse sido lançado comercialmente. A SDL mudou seu nome para Relational Software Inc. (RSI) no ano de 1978 e, em 1979, colocou a primeira versão comercial do software, que foi vendida à base da Força Aérea em Wright-Patterson. Esse foi o primeiro RDBMS comercial no mercado. Depois disso, em 1982, a RSI mudou seu nome para Oracle Systems Corporation (OSC), que posteriormente foi simplificado para Oracle Corporation. Em poucos anos, a Oracle se tornou líder em bancos de dados relacionais.

Uma grande característica deste SGDBR é a integração no mercado da tecnologia da informação, acompanhando as novidades que vêm surgindo, como a inclusão de multimídia, CAD/CAM, gráficos, textos, desenhos, fotografias, som e imagem. Além disso, contempla desenvolvimentos voltados à programação orientada a objetos e banco de dados semânticos. Implementa a linguagem SQL, inclusive incorporando as modificações realizadas no pa-

drão SQL3, a nova versão aprovada pela ANSI, como também sua linguagem própria, voltada a procedimentos, PL/SQL. Seu teor é constituído de vários componentes, como gerador de aplicação, dicionários de dados, gerador de relatório, planilhas eletrônicas, pré-compiladores para linguagens hospedeiras, utilitários, ferramentas de modelagem, gráficos, cálculos estatísticos e armazenagem de dados de vários tipos (data, valores numéricos com e sem decimais, caracteres variáveis ou fixos, binários, imagens, textos, sons).

O SGDBR da Oracle pode ser usado em vários tipos de equipamentos, sendo eles computadores de pequeno e grande porte. Quanto à plataforma, dá suporte a uma gama de ambientes incluindo, UNIX, VMS, MVS, HP, Macintosh, Rede Novell, MS-Windows entre outros. Possui compatibilidade com outros SGDBRs, tais como, DB/2 e SQL/DS da IBM, Sybase, Ingres, Informix, e outros não relacionais como SAS, lipper, Lotus etc. Outras características relacionais são:

- Controle sofisticado de concorrências;
- Suporte total para visões e junções;
- Consistência garantida na leitura de dados;
- Otimização de consultas avançadas baseadas em custo;
- Transações de multietapas garantidas contra falhas;
- Recuperação automática contra quebras ou falhas de computador;
- Indexação dinâmica;
- Pesquisas distribuídas;
- Junções externas (*outer joins* inexistentes em outros SGBDRDs);
- Funções lógicas de textos;
- Usuários e SCHEMAS;
- Índices, *clusters* e *hash clusters*;
- Sequências;

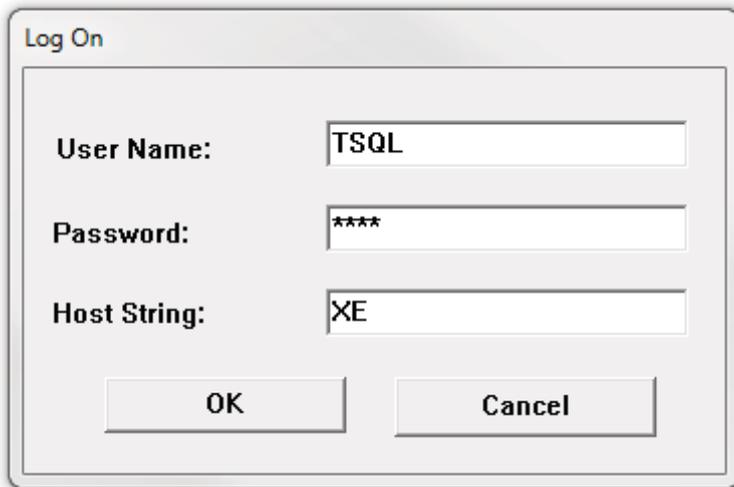
- Procedimentos, pacotes e triggers;
- Sinônimos, papéis e privilégios;
- Segmentos de *Rollback*;
- *Snapshots* e visões materializadas.

No que diz respeito à segurança, o banco de dados Oracle não deixa a desejar, pois o alto grau de desempenho, portabilidade e interoperabilidade o transforma em líder mundial em seu segmento. A integridade e consistência permitem que usuários construam sistemas seguros, confiáveis e com alta disponibilidade de recursos. Algumas características relacionadas à segurança são:

- Suporte multinivelar;
- Integridade de dados;
- Disponibilidade e tolerância a falhas;
- Administração de privilégios;
- Administração da segurança;
- Alto desempenho;
- Suporte a migração;
- Controle de concorrência;
- Suporte de segurança multinivelar a BD distribuído;
- Suporte a padrões internacionais.

Atualmente, a versão do banco de dados encontra-se na 11g, sendo a 9i e 10g as mais difundidas no mercado. A Oracle também disponibiliza outros produtos, que têm como objetivo auxiliar os usuários nas mais diversas áreas de negócio.

3.2 Comunicando com o banco de dados Oracle

A ferramenta **SQL*Plus** acompanha o banco de dados Oracle quando ele é instalado. Ela tem a responsabilidade de fazer a comunicação entre o usuário e o banco de dados. Ao ser acionada, é apresentada uma tela de identificação onde devem ser informados o nome do usuário, senha e a *string* de conexão com o banco de dados com o qual deseja se conectar.

Sua finalidade geral é a execução de comandos SQL, código PL/SQL, procedimentos (*procedures*), gatilhos (*triggers*) e funções (*functions*). A interface desta ferramenta é semelhante ao prompt do MS-DOS, pois disponibiliza um prompt que fica esperando a entrada de uma instrução. Ela também apresenta algumas informações, como, por exemplo, a versão programa e a versão do banco de dados Oracle conectado.

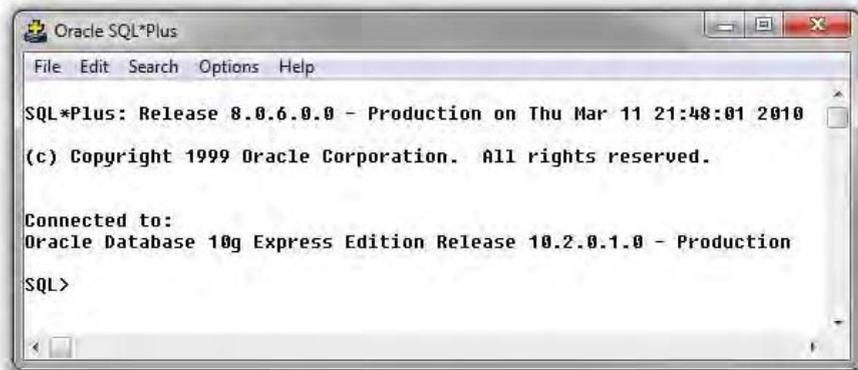



Fig. 3.2: Exemplo da tela do SQL*Plus

Como na maioria das ferramentas, existem diversas configurações de visualização de instruções e resultados, como formatação de linhas, colunas, páginas, comandos para mostrar informações sobre a execução do código, entre outras. Além disso, há uma série de comandos que são úteis para a manipulação destas instruções:

- Editar o comando SQL armazenado no *buffer*;
- Formatar os resultados retornados pelo banco de dados;
- Armazenar os comandos de SQL para disco e recuperá-los para execução;
- Modificar o modo de trabalhar do SQL*Plus;
- Enviar mensagens e receber respostas de outros usuários;
- Listar a definição de qualquer tabela;
- Fazer acesso e copiar dados entre banco de dados.

Na sequência, veja um exemplo de uma instrução digitada e executada do SQL*Plus:

Existem outras ferramentas com interfaces mais amigáveis disponibilizadas pela Oracle para fazer este trabalho com mais rapidez utilizando outros recursos, como, por exemplo, a ferramenta SQL Developer. Outras ferramentas de terceiros que também possuem muitos recursos são SQLTools (Free Software), Toad (Free Software), SQL Navigator e PL/SQL Developer (necessitam de licença).

3.3 Escopo do usuário

Para se ter acesso ao banco de dados Oracle são necessários um usuário e uma senha. Se você não estiver acessando o banco através do usuário administrador do banco de dados, que foi configurado no momento da instalação do banco, o cadastro terá obrigatoriamente que ser feito para que você tenha acesso.

Geralmente é o DBA (*Database Administrator*) quem cria os usuários que

poderão ter acesso ao banco de dados, mas nada impede que você os crie apesar de não ser recomendado quando estamos falando de bancos de produção. Deixe esta tarefa para o DBA. Ele é o administrador do banco e não só cria os usuários como também garante o acesso aos objetos e recursos dos quais precisamos para operar o banco. O papel deste profissional é manter o banco em perfeita ordem e funcionando adequadamente.

Tendo o usuário e senha devidamente habilitados e com permissões para acesso ao banco de dados, basta escolher a ferramenta e efetuar o login.

Ao conectarmos a um banco de dados através de uma das ferramentas mencionadas anteriormente, iniciamos uma sessão dentro dele. Quando solicitamos esta conexão, o banco de dados abre uma sessão onde somente este usuário terá acesso a esta área. Agregado ao usuário também existe uma estrutura chamada de `SCHEMA`, onde todas suas configurações e todos os objetos que foram criados para ele estão disponíveis para acesso, ou seja, esses objetos e configurações têm um dono, que dentro do banco de dados Oracle é chamado de `OWNER`. É óbvio que, se você acabou de criar um usuário, muito provavelmente ele não deverá possuir nenhum objeto criado.

Se um usuário cria um objeto no banco de dados, este objeto pertence a ele. Desta forma, somente ele terá acesso a este objeto, a menos que ele mesmo dê acesso a outros usuários do banco de dados. Sim, isso pode ser feito, e é muito comum de acontecer.

Quando falamos em objetos, referimo-nos a tabelas, packages, triggers, procedures, functions etc. No caso das tabelas que contêm os dados, os usuários com acesso a estes objetos também poderão acessar os dados contidos nelas.

Em grandes sistemas corporativos vemos muito isto acontecer. Por exemplo, temos várias tabelas correspondentes a diversos módulos dentro do sistema. Temos o módulo de RH, o módulo de contas a pagar, contas a receber e assim por diante. Podemos ter criado no banco de dados um usuário para cada módulo e, através deste usuário, dar acessos a estes objetos aos outros usuários (módulos) do sistema ou, ainda, para outro usuário principal, digamos assim, o qual iniciará nosso sistema. Ou seja, podemos ter um único usuário responsável por acessar os objetos que estão criados em outros usuários.

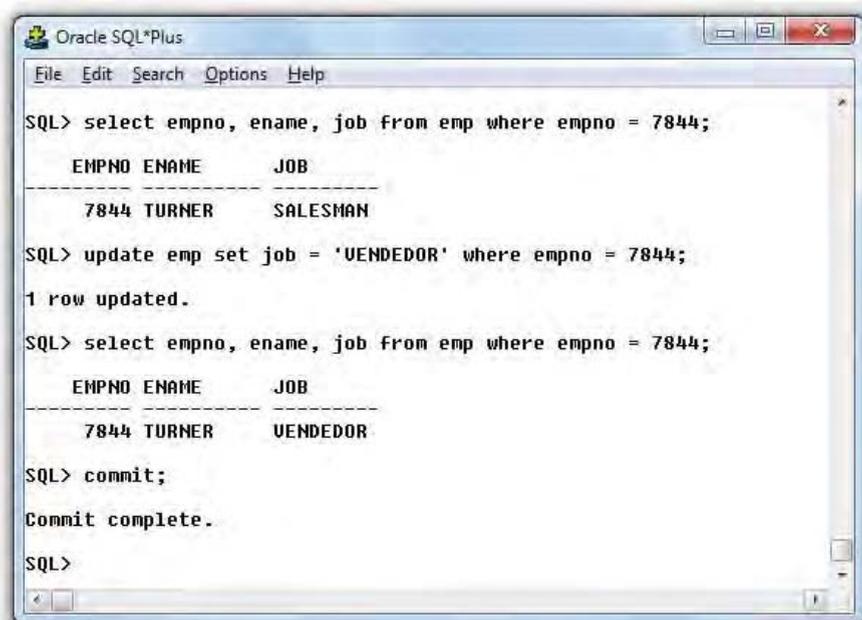
O SGDB Oracle é muito flexível e íntegro quando se trata da manipulação de dados e seus `SCHEMAS` de usuário. Outro ponto interessante é que os dados que disponibilizarmos podem ser restringidos a tal ponto que, se assim quisermos, outros usuários só poderão ter acesso de leitura, não podendo fazer qualquer alteração.

Para que esta manipulação de dados aconteça sem problemas, visando a integridade das informações e garantindo o ótimo funcionamento do banco de dados e programas conectados a ele, o banco Oracle trabalha com transações. No próximo tópico, vamos ver o que significa uma transação.

3.4 Transações

Primeiramente, antes de começarmos a trabalhar com bancos dados Oracle, é preciso conhecer o conceito de transação. Uma das características mais bem-vindas dos bancos de dados Cliente/Servidor, em relação aos bancos de dados desktop, é o conceito de transações. Uma transação é uma unidade lógica de trabalho composta por uma ou mais declarações da *Data Manipulation Language* (DML) ou *Data Definition Language* (DDL). Os comandos para o controle da transação são os comandos necessários para que se possa controlar a efetivação ou não das modificações feitas no banco de dados. Para explicar o que é uma transação, pode-se tomar como exemplo uma transferência bancária na qual um determinado valor é transferido para outra conta. O processo de transferência deste valor consiste em vários passos, que são agrupados de modo que, se não forem concluídos em sua totalidade, ou seja, se a transação não chegar até o final, todas as outras alterações já realizadas serão descartadas e a conta voltará ao seu estado anterior, como se nenhuma transferência tivesse sido realizada. Através deste controle, pode-se garantir que os dados permanecerão consistentes. Os comandos `COMMIT` e `ROLLBACK` da DCL auxiliam neste trabalho.

COMMIT

Torna permanentes todas as alterações feitas no banco de dados durante a sessão. Todas as alterações realizadas em uma determinada transação serão confirmadas caso este comando seja aplicado.

Neste exemplo, é mostrada atualização do cargo referente ao empregado `TURNER`, e logo após, a efetivação da atualização através do comando `COMMIT`.

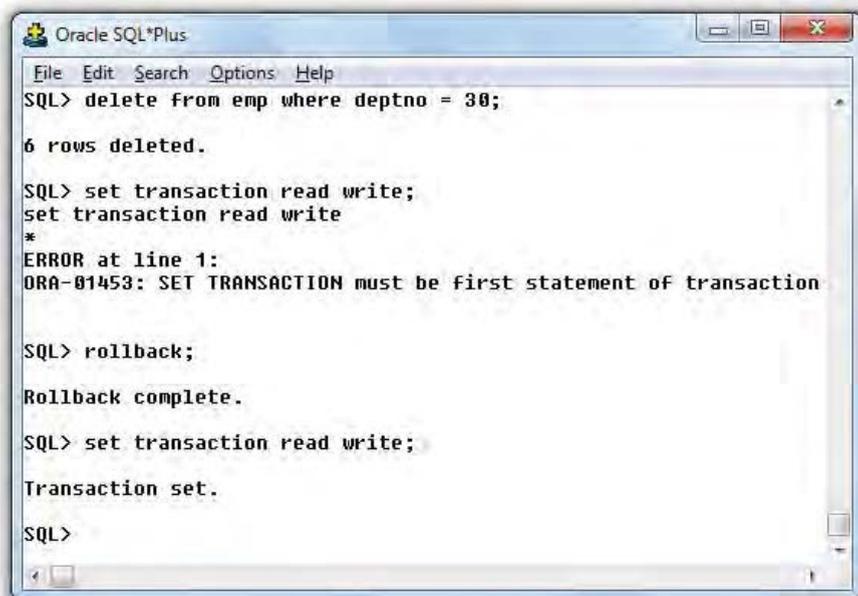
ROLLBACK

Usado para remover todas as alterações feitas desde o último `COMMIT` durante a sessão. Esse comando vai restaurar os dados ao lugar onde eles estavam no último `COMMIT`. Alterações serão desfeitas caso a transação seja encerrada pelo comando `ROLLBACK`.

Já neste exemplo, temos a exclusão de cinco linhas referentes aos empregados que trabalham no departamento 20. Contudo, logo após a exclusão, foi executado um `ROLLBACK`, anulando a ação.

No Oracle, uma transação se inicia com a execução da primeira instrução SQL e termina quando as alterações são salvas ou descartadas. O comando

`SET TRANSACTION` também inicia uma transação – transação explícita. O uso do comando `SET TRANSACTION` determina algumas regras, listadas a seguir.

- Deve ser o primeiro comando da transação (caso contrário, ocorrerá um erro);
- Somente consultas são permitidas na transação;
- Um `COMMIT`, `ROLLBACK` ou qualquer outro comando de DDL (possuem `COMMITS` implícitos) encerram o efeito do comando `SET TRANSACTION`.

3.5 Dicionário de dados do Oracle

O banco de dados Oracle é muito robusto e sua gama de parametrizações, estruturas e arquiteturas requer uma organização para que tudo possa ocorrer de forma harmônica, e para que tanto o banco de dados quanto nós que trabalhamos com ele possamos achar as informações quando necessitamos.

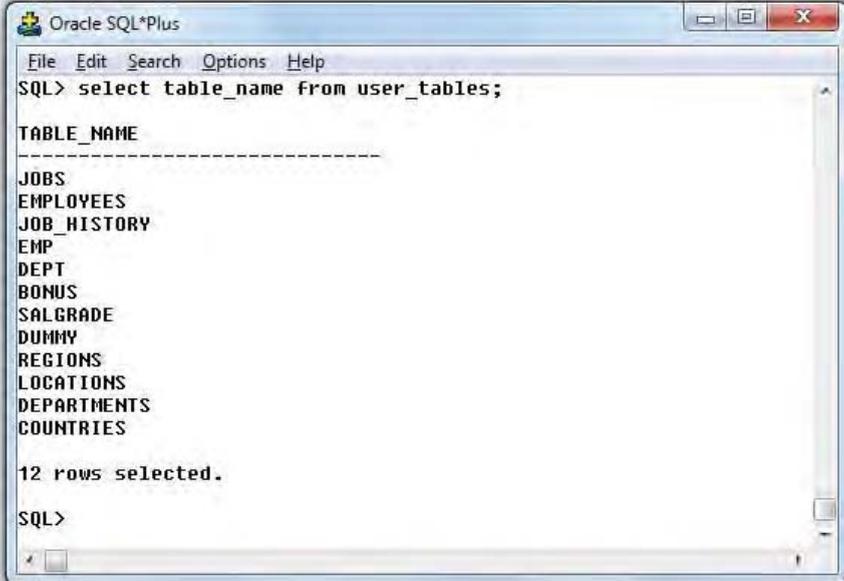
Para que isso aconteça, todas as informações sobre os objetos estão gravadas em tabelas que formam o chamado **dicionário de dados**. Este dicionário é fundamental para que o banco de dados possa operar de forma adequada. A estrutura do dicionário é bastante complexa, por isso existem visões do dicionário com nomes e estruturas mais legíveis para que possamos utilizá-las para recuperar informações pertinentes ao banco de dados. Existem centenas dessas visões. Com toda certeza, trata-se de um SGBD muito completo e estruturado de forma a se adequar aos mais diferentes tipos de negócio.

> **Cuidado!**
>
> As informações no dicionário de dados são mantidas por comandos SQL. Por exemplo, `CREATE TABLE EMPLOYEES` atualiza as tabelas do dicionário, de modo que a visão `USER_TABLES` apresente as informações sobre a nova tabela. Nunca insira ou altere diretamente as tabelas do dicionário: isto invalida o banco de dados, corrompendo totalmente a base de dados. Obviamente não é qualquer usuário que pode alterar o dicionário mas, se você for um DBA, terá acesso total.

Veja algumas das visões mais usadas pelo usuário:

- **DICTIONARY:** mostra todas as visões que compõem o dicionário de dados;
- **USER_TABLES:** mostra as tabelas das quais o usuário é dono;
- **ALL_TABLES:** mostra as tabelas às quais o usuário tem acesso;
- **USER_TAB_COLUMNS:** mostra todas as colunas definidas nas tabelas das quais o usuário é dono;

- **ALL_TAB_COLUMNS:** mostra todas as colunas definidas nas tabelas às quais o usuário tem acesso;
- **ALL_OBJECTS / ALL_SOURCE:** mostra todos os objetos aos quais o usuário tem acesso. Alguns deles são: `TABLES`, `VIEWS`, `PACKAGES`, `PROCEDURES`, `FUNCTIONS`, `CLUSTERS` etc. Os tipos de objeto podem mudar conforme a versão do banco.

Além destas, ainda temos visões com informações sobre índices (tabelas e colunas), `VIEWS`, `CONSTRAINTS`, `TABLESPACES` etc.

Neste exemplo, temos a execução da view `USER_TABLES`, que mostra todas as tabelas pertencentes ao usuário logado.

3.6 Como o Oracle executa comandos SQL

Existem várias formas de se executar declarações DML. O Oracle, através do Optimizer, escolhe a forma mais eficaz para fazer isto. O Optimizer é o programa otimizador do banco de dados Oracle, que é responsável pela melhor escolha possível para executar um comando SQL. Mas não vamos nos ater aqui a explorar suas funcionalidades e capacidades (que são muitas, diga-se de passagem). O que é interessante entender são os passos da execução de um comando SQL.

Quando se executa um comando SQL em um banco de dados, três etapas são executadas, sendo elas: `PARSE`, `EXECUTE` e `FETCH`. Em cada etapa, o Oracle define tarefas que devem ser seguidas para que o resultado seja obtido.

PARSE

A primeira tarefa desta etapa é verificar a sintaxe do comando e validar a semântica.

Depois, é a vez de pesquisar se este comando já foi executado anteriormente, encontrando-se na memória. Se existir, o plano de execução já estará traçado, tendo em vista que o comando já foi analisado e executado antes. Se for constatada a existência, será passado direto para a fase `EXECUTE`, não repetindo a outras ações.

Caso não seja encontrado na memória, o Oracle verifica no dicionário de dados a existência das tabelas e colunas envolvidas no comando, bem como as permissões de acesso a estes objetos.

Após a pesquisa, o otimizador poderá traçar o caminho de acesso para cada tabela presente no comando, montando um plano de execução (também chamado de `PARSE TREE`) para obter e/ou atualizar os dados referentes à requisição feita através deste comando.

Tendo o plano montado, o Oracle o inclui em memória para que a execução seja realizada e para que este plano seja utilizado por outro comando idêntico ao atual.

EXECUTE

A primeira tarefa do `EXECUTE` é aplicar o plano de execução. Com isto, a

leitura dos dados será feita de forma física ou lógica, dependendo da sua localização em disco ou em memória. Se estiver em memória, não será necessário fazer qualquer acesso a disco. Caso contrário, a leitura é feita, e o retorno é armazenado na área de memória do banco de dados.

Já com os dados em memória, será realizada a análise das restrições que consiste em verificar as regras de integridade (`CONSTRAINTS`). Esta tarefa é aplicada a processos de atualização.

Como a anterior, esta tarefa também só é aplicada em processos de atualização, sendo que é preciso bloquear as linhas de dados (`LOCKED`), alocar o segmento de `ROLLBACK`, para que a atualização das linhas seja realizada.

FETCH

Se for o caso em que o comando SQL executado for um `SELECT`, o Oracle retorna as informações.

Capítulo 4

Executando comandos SQL com SQL*Plus

4.1 O que é SQL*Plus?

É uma ferramenta disponibilizada pela Oracle que funciona como um prompt de comando para execução de comandos SQL e PL/SQL. Suas principais funcionalidades são:

- Processamento online de comandos SQL que:
  - Definem a estrutura de base de dados;
  - Realizam consultas *ad-hoc* à base de dados;
  - Manipulam as informações armazenadas na base de dados.

- Formatação de linhas e colunas recuperadas por uma consulta SQL.
- Edição de comandos SQL através de um editor de linhas *built-in* ou de seu editor de textos favorito.
- Armazenamento, recuperação e execução de comandos SQL em/de arquivos.

Na sequência, veremos comandos do SQL*Plus que são utilizados para a manipulação de comandos SQL nesta ferramenta.

4.2 Comandos de edição do SQL*Plus

Os comandos de edição têm o objetivo de modificar a informação armazenada no SQL Buffer. Estes comandos afetam apenas uma linha de cada vez, isto é, a edição é feita linha a linha. Fica guardado no buffer sempre o último comando SQL executado. Os comandos de edição não ficam no Buffer.

Quando listamos o conteúdo do SQL Buffer, o SQL*Plus mostra do lado esquerdo do texto (ao lado da numeração) um asterisco em uma das linhas, indicando que se trata da linha corrente, ou seja, da linha apta para modificação. Desta forma, qualquer comando de edição que viermos a fazer afetará apenas a linha corrente.

L-IST

Este comando tem a finalidade de listar uma ou mais linhas do SQL Buffer. A última linha listada pelo comando se tornará a linha corrente.

```
SQL> list
     select * from DEPT
SQL> l
   select * from DEPT
SQL>
```

A-PPEND

Com este comando, podemos adicionar um trecho de texto ao fim da linha corrente.

```
SQL> a  where rownum < 10
     select * from DEPT where rownum < 10
```

C-HANGE

Este comando de edição tem o objetivo de substituir parte do texto (ou todo) por outro.

Observe que os separadores apresentados são iguais. Podemos utilizar qualquer caractere especial não presente nos textos, porém, dentro de um comando, só podemos usar um deles.

```
SQL> l
     select * from DEPT where rownum < 10
SQL> c/10/20
     select * from DEPT where rownum < 20
```

DEL

Exclui uma determinada linha da área do SQL Buffer. Caso não sejam informados parâmetros que indiquem a linha a ser removida, será excluída a linha corrente.

```
SQL> l
     select *
     from DEPT
     where rownum < 20
SQL>del 3
SQL> l
     select *
     from DEPT
SQL>
```

I-NPUT

Este comando adiciona uma ou mais linhas após a linha corrente (no SQL Buffer). Este comando difere do comando `Append` uma vez que pode voltar ao estado de digitação, abrindo uma nova linha para digitação imediatamente após a linha corrente.

```
SQL> l
     select *
SQL>i from DEPT
SQL> l
     select *
     from DEPT
SQL>
```

ED-IT

Este comando aciona um editor registrado no Windows e passa como parâmetro o nome de um arquivo ou o texto presente no SQL Buffer, de acordo com o comando executado.

Quando omitimos o nome do arquivo a ser editado, o SQL*Plus aciona o editor do sistema passando como parâmetro o texto do SQL Buffer.

Quando desejamos editar um determinado arquivo, seu nome pode ser informado com ou sem a extensão. A extensão default é SQL.

R-UN

Este comando envia o conteúdo do SQL Buffer para o banco de dados e, ao mesmo tempo, apresenta no vídeo as linhas enviadas (lista o SQL Buffer).

```
SQL> r
     select ENAME from EMP where rownum < 3

ENAME
----------
SMITH
ALLEN
```

/ (BARRA)

Quando digitamos uma barra na linha de prompt e em seguida teclamos `Enter`, o SQL*Plus envia o conteúdo do SQL Buffer para o banco de dados, porém não apresenta o texto enviado, isto é, não lista o SQL Buffer. Esta é a diferença entre o comando `Run` e a barra.

```
SQL> /
```

```
ENAME
----------
SMITH
ALLEN
```

EXIT / QUIT

Os comandos `EXIT` e `QUIT` são equivalentes e têm a finalidade de encerrar a sessão do SQL*Plus. Isto significa que a conexão com o banco de dados e, simultaneamente, o próprio programa serão encerrados.

Quando digitamos apenas `QUIT` (ou apenas `EXIT`), o término do programa é considerado normal e a conexão se encerra também normalmente com `COMMIT`.

DESC-RIBE

Este comando tem a finalidade de apresentar a definição de um objeto criado na base de dados Oracle. O objeto pode ser uma `TABLE`, `VIEW`, `SYNONYM`, `FUNCTION`, `PROCEDURE` ou `PACKAGE`.

```
SQL> DESC BONUS
 Name                            Null?    Type
 ------------------------------- -------- ----
 ENAME                                    VARCHAR2(10)
 JOB                                      VARCHAR2(9)
 SAL                                      NUMBER
 COMM                                     NUMBER

SQL>
```

SAVE

Este comando salva o conteúdo do SQL Buffer em um arquivo do sistema operacional.

```
SQL> l
     select ENAME from EMP where rownum < 3
SQL> save EMPREGADOS.sql
```

```
Created file EMPREGADOS.sql
SQL>
```

GET

A recuperação de um texto para o SQL Buffer é feita por meio do comando `GET`. Todo o conteúdo do arquivo é copiado para o SQL Buffer, portanto, este arquivo deve conter apenas um comando.

```
SQL> get EMPREGADOS.sql
     select ENAME from EMP where rownum < 3
SQL>
```

STAR-T / @

Este comando executa o conteúdo de um arquivo de comandos existente no sistema operacional. Cada comando de SQL ou de SQL*Plus é lido e tratado individualmente. Em um arquivo executado por `Start`, podemos incluir diversos comandos de SQL.

O comando `@` é sinônimo de `Start` e o `@@` é semelhante, com a diferença de que, se este comando for incluído em um arquivo de comandos, ele pesquisará o arquivo associado no diretório do arquivo executado e não no diretório local, como seria o caso do `@` e do `Start`.

```
SQL> star EMPREGADOS.sql

ENAME
----------
SMITH
ALLEN

SQL>
```

HOST

Abre um prompt no `DOS`, ou `SHELL` no UNIX, sem fechar o SQL*Plus. Para retornar, basta digitar `EXIT`.

SPOOL

`SPOOL {NOME_ARQUIVO|OFF|OUT}`

Quando queremos armazenar o resultado de uma consulta utilizamos este comando. O resultado é armazenado em um arquivo do sistema operacional que pode, opcionalmente, ser enviado para a impressora padrão, configurada neste sistema operacional. `NOME_ARQUIVO` indica o nome do lugar onde os resultados serão gravados. Ele é gravado com a extensão `LST`, entretanto, pode ser gravado com outra extensão, como por exemplo, `TXT`. Esta mudança pode ocorrer de acordo com o sistema operacional. `OFF` Interrompe a gravação (geração) do arquivo. `OUT` interrompe a geração do arquivo e o envia para impressora.

```
SQL> spool c:\temp\arquivo1.txt
SQL> select ename, job, sal from emp where comm is not null;

ENAME      JOB             SAL
---------- --------- ---------
ALLEN      SALESMAN       1600
WARD       SALESMAN       1250
MARTIN     SALESMAN       1250
TURNER     VENDEDOR       1500

SQL> spool off;
SQL> edit c:\temp\arquivo1.txt

SQL>
```

4.3 Variáveis de sistema

As variáveis de sistema determinam o comportamento da ferramenta SQL*Plus e a apresentação dos resultados exibidos por ela. Através da própria ferramenta, é possível consultar estas variáveis e também configurá-las. Usamos o comando `SET` para configurar as variáveis, e `SHOW` para visualizar seus valores.

- **ARRAY[SIZE] {15|n}** Define o número de linhas que o SQL*Plus deve recuperar do banco de dados a cada leitura (*Fetch*). Vale ressaltar que

não se trata do aspecto visual em tela, mas sim, do acesso interno aos dados do banco, pela ferramenta.

- **DEF[INE] {'&'|c|OFF|ON}** Determina o caractere a ser usado para prefixar variáveis de substituição. `ON` ou `OFF` controla se o SQL*Plus irá ou não pesquisar no texto a procura de variáveis de substituição para substituí-las por valores. Ao usarmos `ON`, o prefixo de substituição, por padrão, será `&`.

  Exemplo:

```
SQL> set define on
SQL> select ename, job, sal from emp where deptno = &deptno
    /
Enter value for deptno: 20
old   1: select ename, job, sal from emp where deptno = &deptno
new   1: select ename, job, sal from emp where deptno = 20

ENAME      JOB              SAL
---------- --------- ---------
SMITH      CLERK            800
JONES      MANAGER       3272.5
SCOTT      ANALYST         3000
ADAMS      CLERK           1100
FORD       ANALYST         3000

SQL> set define off
SQL> select ename, job, sal from emp where deptno = &deptno;
Bind variable "DEPTNO" not declared.
SQL>
```

  Você também pode inserir uma descrição mais inteligível para solicitar os dados para as variáveis de substituição, utilizando `ACCEPT`. Para isto, salve seu `SELECT` em um arquivo e, antes do comando, coloque as seguintes linhas, como no exemplo a seguir.

```
set echo off
accept qtd prompt "Informe a quantidade de empregados desejada:"
select ename
```

```
      ,sal
from   emp
where  sal is not null
and    rownum < &qtd
order  by 2 desc
/
```

Agora é só executar o arquivo pelo SQL*PLUS, utilizando o comando `@`, por exemplo.

- **Variável de Substituição** Você também pode utilizar uma variável de substituição, por exemplo, o E comercial, de forma dupla para reutilizar o valor da variável sem ter que solicitá-lo sempre ao usuário. Veja o exemplo a seguir:

```
SQL> select employee_id, last_name, job_id, &&column_name
  2  from   employees
  3  where  job_id = 'SA_MAN'
  4  order  by &column_name
  5  /
Enter value for column_name: salary
old   1: select employee_id, last_name, job_id, &&column_name
new   1: select employee_id, last_name, job_id, salary
old   4: order  by &column_name
new   4: order  by salary

EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        149 Zlotkey                   SA_MAN       15373.05
        148 Cambrault                 SA_MAN        16105.1
        147 Errazuriz                 SA_MAN        17569.2
        146 Partners                  SA_MAN       19765.35
        145 Russell                   SA_MAN        20497.4

SQL>
```

- **ECHO {OFF|ON}** Quando executamos comandos SQL a partir de um arquivo, podemos definir se eles devem ser exibidos na tela (`ON`) ou não (`OFF`).

Exemplo:

```
SQL>edit c:\temp\arquivo3.txt

SQL> get c:\temp\arquivo3.txt
  1* select ename, job, sal from emp where deptno = &deptno
SQL>
SQL> @c:\temp\arquivo3.txt
Bind variable "DEPTNO" not declared.
SQL> set define on
SQL> @c:\temp\arquivo3.txt
Enter value for deptno: 20
old   1: select ename, job, sal from emp where deptno = &deptno
new   1: select ename, job, sal from emp where deptno = 20

ENAME      JOB              SAL
---------- --------- ---------
SMITH      CLERK            800
JONES      MANAGER       3272.5
SCOTT      ANALYST         3000
ADAMS      CLERK           1100
FORD       ANALYST         3000

SQL> set echo on
SQL> @c:\temp\arquivo3.txt
SQL> select ename, job, sal from emp where deptno = &deptno
  /
Enter value for deptno: 20
old   1: select ename, job, sal from emp where deptno = &deptno
new   1: select ename, job, sal from emp where deptno = 20

ENAME      JOB              SAL
---------- --------- ---------
SMITH      CLERK            800
JONES      MANAGER       3272.5
SCOTT      ANALYST         3000
ADAMS      CLERK           1100
FORD       ANALYST         3000
```

```
SQL>
```

- **HEA[DING] {ON | OFF}** Determina se os cabeçalhos de coluna devem ser apresentados (default: `ON`).

  Exemplo:

```
SQL> select ename, job, sal from emp where comm is not null;

ENAME      JOB              SAL
---------- --------- ---------
ALLEN      SALESMAN       1600
WARD       SALESMAN       1250
MARTIN     SALESMAN       1250
TURNER     VENDEDOR       1500

SQL> set heading off
SQL> /

ALLEN      SALESMAN       1600
WARD       SALESMAN       1250
MARTIN     SALESMAN       1250
TURNER     VENDEDOR       1500

SQL> set heading on
SQL> /

ENAME      JOB              SAL
---------- --------- ---------
ALLEN      SALESMAN       1600
WARD       SALESMAN       1250
MARTIN     SALESMAN       1250
TURNER     VENDEDOR       1500

SQL>
```

- **LIN[ESIZE] {80 | n}** Indica o número de caracteres por linha (default: `80`).

  Exemplo:

```
SQL> set lines 10
SQL> select ename, job, sal from emp where comm is not null;

ENAME
----------
JOB
---------
      SAL
---------
ALLEN
SALESMAN
     1600

WARD
SALESMAN
     1250

MARTIN
SALESMAN
     1250

TURNER
VENDEDOR
     1500


SQL> set lines 80
SQL> select ename, job, sal from emp where comm is not null;

ENAME      JOB             SAL
---------- --------- ---------
ALLEN      SALESMAN       1600
WARD       SALESMAN       1250
MARTIN     SALESMAN       1250
TURNER     VENDEDOR       1500

SQL>
```

- **PAGES[IZE] {24 | n}** Indica o número de linhas para cada página do

relatório (default: 24).

Exemplo:

```
SQL> set pages 10
SQL> select empno, ename, job, mgr from emp;

    EMPNO ENAME      JOB              MGR
--------- ---------- --------- ---------
     7369 SMITH      CLERK           7902
     7499 ALLEN      SALESMAN        7698
     7521 WARD       SALESMAN        7698
     7566 JONES      MANAGER         7839
     7654 MARTIN     SALESMAN        7698
     7698 BLAKE      MANAGER         7839
     7782 CLARK      MANAGER         7839

    EMPNO ENAME      JOB              MGR
--------- ---------- --------- ---------
     7788 SCOTT      ANALYST         7566
     7839 KING       PRESIDENT
     7844 TURNER     VENDEDOR        7698
     7876 ADAMS      CLERK           7788
     7900 JAMES      CLERK           7698
     7902 FORD       ANALYST         7566
     7934 MILLER     CLERK           7782

14 rows selected.

SQL>
```

- **TIMI[NG] {OFF|ON}** Controla a apresentação de estatísticas de tempo. ON mostra as estatísticas em cada comando SQL ou bloco de PL/SQL.

  Exemplo:

```
SQL> set timing on
SQL> select * from dept;
```

```
   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
50 TI             BRASIL

real: 83
SQL>
```

- **PAU[SE] <mensagem>** Mostra a mensagem após o preenchimento da página de tela.

  Exemplo

  ```
  SQL> set pause 'tecle enter para continuar...'
  SQL>
  ```

- **PAU[SE] {ON | OFF}** Indica se o mecanismo de pausa após o preenchimento de uma página deve ser acionado (default: `OFF`).

  Exemplo:

  ```
  SQL> set pause on
  SQL> select empno, ename, job, mgr from emp;
  tecle enter para continuar...

      EMPNO ENAME      JOB             MGR
  --------- ---------- --------- ---------
       7369 SMITH      CLERK          7902
       7499 ALLEN      SALESMAN       7698
       7521 WARD       SALESMAN       7698
       7566 JONES      MANAGER        7839
       7654 MARTIN     SALESMAN       7698
       7698 BLAKE      MANAGER        7839
       7782 CLARK      MANAGER        7839
       7788 SCOTT      ANALYST        7566
       7839 KING       PRESIDENT
       7844 TURNER     VENDEDOR       7698
  ```

```
      7876 ADAMS      CLERK          7788
      7900 JAMES      CLERK          7698
      7902 FORD       ANALYST        7566
      7934 MILLER     CLERK          7782

14 rows selected.

SQL>
```

- **SERVEROUT[PUT] {OFF|ON} [SIZE n] [FOR[MAT] {WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}]** Controla a apresentação das informações geradas através do pacote `DBMS_OUTPUT` em `stored procedures` ou blocos de PL/SQL. O parâmetro `Size` determina o número de bytes que podem ser buferizados dentro do Oracle (o valor deve variar entre 2000 e 1.000.000). O default é 2000. Quando `WRAPPED` é habilitado, a quebra de linha ocorre a cada `LINESIZE` caracteres. Quando `WORD_WRAPPED` é utilizado, a quebra de linhas ocorre a cada `LINESIZE` caracteres, porém em final de palavra. Quando `TRUNCATED` é habilitado, cada linha é truncada em `LINESIZE` caracteres.

  Exemplo:

```
SQL> declare
  2    x varchar2(100);
  3  begin
  4    x := 'Curso de SQL';
  5    dbms_output.put_line(x);
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL> set serveroutput on
SQL> /
Curso de SQL

PL/SQL procedure successfully completed.
```

```
SQL>
```

- **ESC[APE] {\|c|OFF|ON}** Define qual o caractere a ser usado como escape. `ON` altera o valor `<c>` para o default `\`. Este caractere é utilizado antes do caractere para variáveis de substituição para indicar que o texto a seguir deve ser considerado normal e não uma substituição.

  Exemplo:

```
SQL> show escape
escape OFF
SQL> set escape on
SQL> show escape
escape "\" (hex 5c)
SQL> select 'SQL &Cia' from dual;
Enter value for cia:
old   1: select 'SQL & Cia' from dual
new   1: select 'SQL ' from dual

'SQL
----
SQL

SQL> select 'SQL \& Cia' from dual;

'SQL&CIA'
---------
SQL & Cia

SQL>
```

- **SQLT[ERMINATOR] { ;|c|OFF|ON}** Indica qual caractere o SQL*PLUS reconhecerá como fim de linha de execução. `OFF` indica que não existe caractere associado, o fim do comando é reconhecido por uma linha inteira em branco. A opção `ON` retorna ao valor default de `;`.

  Exemplo:

```
SQL> show sqlterminator;
sqlterminator ";" (hex 3b)
SQL> set sqlterminator '@'
SQL> select * from departamento@

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
50 TI                 BRASIL

SQL> set sqlterminator ';'
SQL> select * from departamento;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
       50 TI             BRASIL

SQL>
```

- **SQLP[ROMPT] {SQL>|text }** Indica o texto de prompt para o SQL*PLUS.

  Exemplo:

```
SQL> show sqlprompt
sqlprompt "SQL> "
SQL>
SQL> set sqlprompt "CURSO SQL>"
CURSO SQL>
CURSO SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
```

```
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
50 TI             BRASIL

CURSO SQL> set sqlprompt "SQL>"
SQL>
SQL>
SQL>
```

Modificar a forma como o SQL*Plus mostra seu prompt:

```
SQL> set sqlprompt "&&_USER@&&_CONNECT_IDENTIFIER SQL>"
Enter value for _user: tsql
Enter value for _connect_identifier: xe2
tsql@xe2 SQL>
tsql@xe2 SQL>
tsql@xe2 SQL> set sqlprompt "&_USER@&_CONNECT_IDENTIFIER SQL>"
tsql@xe2 SQL>
tsql@xe2 SQL>
```

- **TERMOUT (ON|OFF)** Exibe na tela os resultados de comandos executados a partir de arquivos, como, por exemplo, `PROMPTs`.

  Exemplo:

```
SQL> set termout off
SQL> @"C:\base_HR\hr_main.sql"
SQL>

SQL> set termout on
SQL> @"C:\base_HR\hr_main.sql"

Preparando execução...espere

---------------------------------
Executando arquivo HR_DROP.SQL ....

=> Excluindo procedimento ADD_JOB_HISTORY ....
=> Excluindo procedimento SECURE_DML ....
=> Excluindo tabela COUNTRIES ....
```

```
...

--------------------------------
Executando arquivo HR_CRE.SQL ....

=> Criando tabela REGIONS ....
=> Criando tabela COUNTRIES ....
=> Criando tabela LOCATIONS ....
=> Criando visão EMP_DETAILS_VIEW ...

---------------------------------
...

Execução finalizada...
```

- **FEEDBACK (n|ON|OFF)** Exibe o retorno de um comando SQL. No caso de um comando `SELECT`, exibe o número de registros que o comando retornou, por exemplo: "4 linhas selecionadas". Já um comando `CREATE`, retorna se o objeto em questão foi ou não criado.

  O valor default para `FEEDBACK` é `6`. Desta forma, somente a partir de 6 ocorrências no resultado é que ele mostrará o retorno. No caso de um comando `SELECT`, se o resultado for o retorno de 6 ou mais registros, aparecerá a quantidade retornada. Caso contrário, não. No caso de um comando criando uma tabela, por exemplo, se o valor configurado para o `FEEDBACK` também for 6, ele não mostrará o resultado dizendo ou não se criou, pois, para este comando, o retorno é apenas uma linha.

  `FEEDBACK 0` é o mesmo que `OFF`. `FEEDBACK ON` ativa o comando e assume o último número definido para ele.

  Exemplo:

```
SQL> set feedback 5
SQL> select ename from emp where comm is not null;

ENAME
----------
ALLEN
```

```
WARD
MARTIN
TURNER

SQL>

SQL> set feedback 3
SQL> select ename from emp where comm is not null;

ENAME
----------
ALLEN
WARD
MARTIN
TURNER

4 linhas selecionadas.

SQL>

SQL> set feedback 3
SQL> select ename from emp where comm is not null;

ENAME
----------
ALLEN
WARD
MARTIN
TURNER
```

4.4 Definindo Variáveis no SQL*Plus

A variável de sistema DEFINE, que é configurada através do comando SET, serve para ativar ou não as definições referentes ao uso das variáveis de substituição. Contudo, existe também um comando com este mesmo nome que é utilizado para definir variáveis no SQL*Plus, as quais serão usadas como variáveis de substituição. Veja o exemplo abaixo:

```
SQL> define employee_num = 200
SQL> select employee_id, last_name, salary, department_id
  2  from   employees
  3  where  employee_id = &employee_num;

EMPLOYEE_ID LAST_NAME                     SALARY DEPARTMENT_ID
----------- ------------------------- ---------- -------------
        200 Whalen                       6442,04            10

SQL> undefine employee_num
SQL>
```

Neste exemplo, está sendo definida uma variável chamada EMPLOYEE_NUM. Note que além de ser definida, a mesma está recebendo um valor por padrão. Depois de definida, ou melhor, declarada, a variável é utilizada como uma variável de substituição, sendo precedida por E comercial. Para limpar a definição, utilize o comando UNDEFINE.

4.5 Configurações iniciais de login do SQL*Plus

O SQL*Plus possui um arquivo de configuração que é acionado cada vez que o programa é iniciado. Neste arquivo, é possível inserir configurações para que, ao iniciar o programa, elas sejam executadas. Configurações como o número de linhas por página (`PAGESIZE`) e caracteres por linha (`LINESIZE`) podem ser inseridas neste arquivo. Para inserir as configurações, basta abrir o arquivo `glogin.sql` e inseri-las de forma bem parecida como se estivesse digitando no próprio programa. Veja um exemplo em:

C:\oraclexe\app\oracle\product`.2.0\server\sqlplus\admin

No corpo do arquivo, temos:

```
--
-- Copyright (c) 1988, 2004, Oracle Corporation.
-- All Rights Reserved.
--
-- NAME
--   glogin.sql
```

```
--
-- DESCRIPTION
--   SQL*Plus global login "site profile" file
--
--   Add any SQL*Plus commands here that are to be executed when
--   a user starts SQL*Plus, or uses the SQL*Plus CONNECT
--   command
--
-- USAGE
--   This script is automatically run
--

-- Used by Trusted Oracle
COLUMN ROWLABEL FORMAT A15

-- Used for the SHOW ERRORS command
COLUMN LINE/COL FORMAT A8
COLUMN ERROR    FORMAT A65  WORD_WRAPPED

-- Used for the SHOW SGA command
COLUMN name_col_plus_show_sga FORMAT a24
COLUMN units_col_plus_show_sga FORMAT a15
-- Defaults for SHOW PARAMETERS
COLUMN name_col_plus_show_param FORMAT a36 HEADING NAME
COLUMN value_col_plus_show_param FORMAT a30 HEADING VALUE

-- Defaults for SHOW RECYCLEBIN
COLUMN origname_plus_show_recyc   FORMAT a16 HEADING
                                              'ORIGINAL NAME'
COLUMN objectname_plus_show_recyc FORMAT a30 HEADING
                                              'RECYCLEBIN NAME'
COLUMN objtype_plus_show_recyc    FORMAT a12 HEADING
                                              'OBJECT TYPE'
COLUMN droptime_plus_show_recyc   FORMAT a19 HEADING 'DROP TIME'

-- Defaults for SET AUTOTRACE EXPLAIN report
-- These column definitions are only used when SQL*Plus
-- is connected to Oracle 9.2 or earlier.
COLUMN id_plus_exp FORMAT 990 HEADING i
```

```
COLUMN parent_id_plus_exp FORMAT 990 HEADING p
COLUMN plan_plus_exp FORMAT a60
COLUMN object_node_plus_exp FORMAT a8
COLUMN other_tag_plus_exp FORMAT a29
COLUMN other_plus_exp FORMAT a44

-- Default for XQUERY
COLUMN result_plus_xquery HEADING 'Result Sequence'

-- Banco Curso SQL
SET PAGESIZE 24
SET LINES 80
```

4.6 Verificando Variáveis de Substituição no SQL*Plus

Para verificar se as substituições referentes as variáveis de substituição foram feitas corretamente na instrução SQL, use o comando VERIFY. O comando O comando SET VERIFY ON faz com que o SQL*Plus exiba o texto de um comando após ele substituir as variáveis de substituição pelos respectivos valores informados. Vamos a um exemplo:

```
SQL> set verify on
SQL> select employee_id, last_name, salary
  2  from   employees
  3  where  employee_id = &employee_num
  4  /
Enter value for employee_num: 200
old   3: where  employee_id = &employee_num
new   3: where  employee_id = 200

EMPLOYEE_ID LAST_NAME                     SALARY
----------- ------------------------- ----------
        200 Whalen                       6442.04

SQL>
```

Note no exemplo, que é mostrado o texto OLD e NEW, com o antes e depois da substituição das variáveis.

4.7 Automatizar login do SQL*Plus

Quando executamos o SQL*Plus, é solicitado um usuário, senha e string do banco. Podemos fazer com que, ao iniciar, ele já conecte automaticamente ao banco de dados. Clique com o botão direito do mouse no atalho do SQL*Plus e vá em `Propriedades`. No campo `Destino` na aba `Atalho`, logo após a chamada do `.exe`, coloque <usuário>/<senha>@<string_conexão>. Por exemplo: `tsql/tsql@xe2`. Veja o print a seguir.

[image images/img_pg_50_do_pdf.png]

Capítulo 5

Limites do SGDB Oracle

A seguir, os limites definidos para os diversos componentes que constituem ou que são utilizados dentro da plataforma do banco de dados ou nas suas linguagens.

- **Tabelas:** o número máximo de tabelas *clusterizadas* é 32. E é ilimitado por banco de dados.
- **Colunas:** o número máximo por tabela é de 1000 colunas. Por índice (ou índice clusterizado), é de 32 colunas. Por índice bitmap, é de 30 colunas.
- **Linhas:** o número de linhas por tabela é ilimitado.
- **Índices:** o número de índices por tabela é ilimitado. Por tamanho de colunas indexadas, é de 75% do tamanho total do bloco do banco de dados (*minus some overhead*).

- **Caracteres:** a quantidade para nomes de objetos é limitada a 30 caracteres.
- **Cláusula GROUP BY:** a expressão *GROUP BY* e todas as expressões de agregação (por exemplo, SUM, AVG) devem estar dentro de um único bloco na base de dados.
- **CONSTRAINTS:** o número de *constraints* por coluna é ilimitado.
- **SUBQUERIES:** o nível de *subqueries*, em um comando SQL, é ilimitado na cláusula `FROM` quando no primeiro nível de uma *query*, e é limitado a 255 na cláusula `WHERE`.
- **Grupo de valores:** o campo `data` pertence ao intervalo de 1-Jan-4712BC a 31-Dec-4712AD.
- **Tabelas ou VIEWS "joined" em uma QUERY:** ilimitado.
- **PACKAGES armazenadas:** em PL/SQL e Developer/2000 pode haver limites no tamanho de `PROCEDURES` armazenadas que elas podem chamar. O limite em geral está entre 2000 a 3000 linhas de código.
- **TRIGGERS em cascata:** depende do sistema operacional; geralmente 32.
- **Usuários e regras:** é limitado a 2, 147, 483, 638.
- **Partições:** comprimento máximo de linha da chave particionada é de 4KB – *overhead*. O número máximo de colunas na chave particionada é de 16. O número máximo de partições disponíveis por tabela ou índice é de 64KB – 1 partição.
- **Definição para CREATE MATERIALIZED VIEW:** tem tamanho máximo de 64K bytes.

O limite de até quanto uma instrução SQL pode retornar (resposta) depende de alguns fatores, incluindo configurações, espaço em disco e memória.

Quando um objeto está instanciado na memória, não existe um limite fixo para o número de atributos neste objeto. Mas o total máximo do tamanho da memória para um objeto instanciado é de 4GB. Quando um objeto instanciado é inserido dentro da tabela, os atributos são alocados em colunas separadas, e o limite de 1000 colunas é aplicado.

5.1 Tipos de dados do SGDB Oracle

Na sequência, veja os tipos de dados utilizados pelas linguagens SQL e PLSQL para a construção de programas ou parametrizações do banco de dados.

VARCHAR2

O tamanho máximo para campos de tabela é de 4000 bytes. O tamanho máximo para PL/SQL é de 32.767 bytes. O `VARCHAR2` é variável e somente usa o espaço que está ocupado. Diferentemente do `CHAR`.

Comentário: `VARCHAR` é um subtipo (assim como `STRING`) que existe por questões de compatibilidade com outras marcas de banco de dados e também com o padrão SQL. Entretanto, a Oracle, no momento, não recomenda o uso do tipo `VARCHAR` porque sua definição deve mudar à medida que o padrão SQL evoluir. Deve-se usar `VARCHAR2`.

CHAR

O tamanho máximo é de 2000 bytes.

Comentário: O tipo `CHAR` é usado para conter dados de string de comprimento fixo. Ao contrário das strings de `VARCHAR2`, uma string `CHAR` sempre contém o número máximo de caracteres.

Outros tipos

- **NCHAR**: tamanho máximo de 2000 bytes

- **NCHAR VARYING**: tamanho máximo de 4000 bytes
- **CHAR VARYING**: tamanho máximo de 4000 bytes

NUMBER(p,s)

É um numero com sinal e ponto decimal, sendo: (p)recisão de 1 a 38 dígitos (e)scala -84 a 127.

Este tipo também possui subtipos como:

- **DECIMAL:** igual a NUMBER;
- **DEC:** igual a DECIMAL;
- **DOUBLE PRECISION:** igual a NUMBER;
- **NUMERIC:** igual a NUMBER;
- **INTEGER:** equivalente a NUMBER(38);
- **INT:** igual a INTEGER;
- **SMALLINT:** igual a NUMBER(38);
- **FLOAT:** igual a NUMBER;
- **FLOAT(prec):** igual a NUMBER(prec), mas a precisão é expressa em termos de bits binários, não de dígitos decimais. A precisão binária pode variar de 1 até 126.

DATE

O tipo `DATE` é usado para armazenar os valores de data e hora. Um nome melhor seria DATETIME porque o componente de hora está sempre lá, independente de você usá-lo ou não. Se não for especificada a hora ao atribuir um valor para uma variável do tipo `DATE`, o padrão de meia-noite (12:00:00 a.m.) será usado.

Ele é do tipo: 1 JAN 4712 BC até 31 DEC 4712 AD (DATA com hora, minuto e segundo).

TIMESTAMP

Data com segundos fracionais. Permite o armazenamento de dados de tempo, como o ano, o mês, a hora, minuto e o segundo do tipo de dados `DATE`, além dos valores de segundos fracionais. Existem muitas variações desse tipo de dados, como `WITH TIMEZONE`, `WITH LOCALTIMEZONE`.

INTERVAL YEAR TO MONTH

Armazenados com um intervalo de anos e meses. Permite que os dados de tempo sejam armazenados como um intervalo de anos e meses. Usado para representar a diferença entre dois valores de data/hora em que apenas o ano e o mês são significativos.

INTERVAL DAY TO SECOND

Armazenados como um intervalo de dias, horas, minutos e segundos. Permite que os dados de tempo sejam armazenados como um intervalo de dias, horas, minutos e segundos. Usado para representar a diferença precisa entre dois valores de data/hora.

BOOLEAN

É do tipo `True` e `False`.

LONG

Somente pode existir uma coluna por tabela. Seu tamanho máximo para tabela é de 2 GB. Seu tamanho máximo para PL/SQL é de 32.760.

RAW

O tamanho máximo para campos de tabela é de 2000 bytes. Seu tamanho máximo para PL/SQL é de 32.767.

`LONG RAW` é outro tipo parecido com `RAW`, a diferença é que ele possui 7 bytes a menos quando utilizado em PL/SQL.

Campos LONG

Existem algumas restrições para campos `LONG` e `LONG RAW`.

- Não se pode criar um `OBJECT TYPE` com o atributo de `LONG`.
- Uma coluna `LONG` não pode estar dentro da cláusula `WHERE` ou com referência integral dos dados, exceto `NULL` ou `NOT NULL`.
- Uma função não pode retornar um campo `LONG`.
- Uma tabela poderá ter somente um campo `LONG`.
- `LONG` não pode ser indexada.
- `LONG` não pode usar cláusulas além do `WHERE`, já mencionado, `GROUP BY`, `ORDER BY` e `CONNECT BY`.

Uma dica para você usar um campo `LONG` na cláusula `WHERE` é criar uma tabela temporária com os campos da tabela original, mas alterando um tipo `LONG` para `CLOB`. Também é possível alterar diretamente na tabela o campo `LONG` para `CLOB`, caso não tenha problema de alterar a estrutura da tabela original. Veja o exemplo:

```
create global temporary table table_temp (campo_lob clob) on
commit preserve rows
/

insert into table_temp select to_lob(coluna_lob) from
table_original
/
```

Outra alternativa, caso você tenha conhecimentos em PLSQL, é a criação de um programa para realizar a pesquisa. Com recursos da PLSQL você consegue utilizar funções que não são permitidas, pelo menos nestes casos, em SQL. O código pode ser parecido com o mostrado a seguir.

```
declare
  cursor c1 is
  select view_name, text
```

```
  from user_views;
begin
  for r1 in c1 loop
    if r1.text like '%emp%' then
      dbms_output.put_line(r1.view_name);
    end if;
  end loop;
end;
```

> Por existirem estas limitações, a Oracle criou a família LOB (CLOB, BLOB, NLOB, BFILE), que visa resolver alguns destes problemas enfrentados com o tipo LONG.

CLOB

O tamanho máximo é (4 GB - 1) * DB_BLOCK_SIZE, parâmetro de inicialização (8TB a 128TB). O número de colunas CLOB por tabela é limitado somente pelo número máximo de colunas por tabela. Ele armazena textos. Estes textos são validados conforme o caracter set, ou seja, armazena acentuação etc. Tipos LOB surgiram em substituição aos tipos LONG e LONG RAW, pois estes só permitiam uma coluna por tabela. Os tipos LOB permitem mais de uma coluna.

NCLOB

O tamanho máximo é de (4 GB - 1) * DB_BLOCK_SIZE, parâmetro de inicialização (8TB a 128TB). O número de colunas NCLOB por tabela é limitado somente pelo número máximo de colunas por tabela. Objeto de grande capacidade de caracteres nacionais – contém até 4GB de caracteres de bytes simples ou caracteres multibyte que atendem o conjunto de caracteres nacional definido pelo banco de dados Oracle.

BLOB

O tamanho máximo é de (4 GB - 1) * DB_BLOCK_SIZE, parâmetro de inicialização (8TB a 128TB).

O número de colunas BLOB por tabela é limitado somente pelo número máximo de colunas por tabela. Armazenam dados não estruturados como: som, imagem, dados binários.

BFILE

O tamanho máximo é de 4GB. O tamanho máximo para o nome do arquivo é de 255 caracteres. O tamanho máximo para o nome do diretório é de 30 caracteres. O número máximo de BFILEs abertos é limitado pelo valor do parâmetro de inicialização SESSION_MAX_OPEN_FILES, o qual é limitado pelo número máximo de arquivos abertos que o sistema operacional suporta.

ROWID

É um tipo especial usado para armazenar os ROWIDs (endereços físicos) das linhas armazenadas em uma tabela.

5.2 Resumo

Em resumo, os tipos comumente utilizados são: CHAR, VARCHAR2, NUMBER, DATE, BOOLEAN e os da família LOB.

Capítulo 6

Gerenciando usuários

6.1 Manipulando usuários

Vimos anteriormente que para se conectar ao banco de dados é necessário ter um usuário e uma senha. O escopo de criação de usuários geralmente faz parte da função do Administrador do Sistema, também chamado de DBA. Ele é quem cria os usuários dentro do banco, concedendo assim, o acesso ao ambiente.

Entretanto, é importante saber como criar um usuário e conceder privilégios para que possa operar no sistema do banco de dados. Note que, para criar um usuário, é necessário estar conectado ao banco. Para isto, temos que ter um usuário que já esteja criado e que tenha poder para criar outros usuários.

Quando um banco de dados Oracle é instalado, existem dois usuários que são criados automaticamente. Um deles é o `SYS` e outro o `SYSTEM`. As senhas para estes usuários podem ser modificadas também no momento da

instalação do banco de dados. Caso contrário, as senhas padrões são: SYS e MANAGER, respectivamente.

Além das definições de nome e senha, outras informações também podem ser definidas, como em quais TABLESPACE e TABLESPACE TEMPORARY o usuário vai ser alocado, quais as quotas, ou seja, quanto de espaço nestas TABLESPACES ele terá para a criação de objetos e manipulação de dados ou se ele vai ser agregado a algum PROFILE específico. Mas como foi dito, estas informações cabem aos DBA definirem. Quando criamos um usuário, não precisamos informar todas estas definições. O Oracle trabalha com valores padrões e, caso não sejam informados, ele usará estes padrões para estabelecer estas configurações.

Como já foi mencionado anteriormente, ao criar um usuário, o Oracle também cria um SCHEMA, onde estarão agregados todos os objetos que ele venha a criar. Desta forma, quando outros usuários quiserem se utilizar de objetos dos quais eles não sejam os donos, terão que usar o nome deste SCHEMA na frente do nome do objeto. Exemplo: TSQL.FILME. Mas lembre-se: mesmo utilizando o nome do esquema na frente do objeto, outro usuário só terá este acesso caso o proprietário do objeto tenha dado concessão de acesso (GRANT). O DBA também pode estabelecer estas configurações e definir que, a cada novo usuário criado, o mesmo assuma estas diretrizes. Desta forma, os usuários que irão criar objetos ou manipulá-los não precisarão se preocupar com esses detalhes.

Vejamos a criação básica de um usuário:

```
SQL> create user TSQL identified by teste123;

User created.

SQL>
```

As informações definidas no momento da criação do usuário podem ser alteradas posteriormente. Para isso, é utilizado o comando ALTER USER. No exemplo a seguir, vamos alterar a senha do usuário que acabamos de criar.

```
SQL> alter user TSQL identified by TSQL;
```

```
User altered.

SQL>
```

Para excluir um usuário, usamos o comando DROP ou DROP CASCADE. Este último é utilizado quando o usuário é proprietário de algum objeto. Vale ressaltar que um usuário conectado não pode ser excluído.

```
SQL> drop user TSQL;
drop user TSQL
*
ERROR at line 1:
ORA-01922: CASCADE must be specified to drop 'TSQL'


SQL>
```

Este erro ocorreu pelo fato de que o usuário possui objetos criados em seu esquema. Nesses casos, para excluí-lo, devemos utilizar a expressão CASCADE. Veja de novo a execução, agora utilizando CASCADE:

```
SQL> drop user TSQL cascade;

User dropped.

SQL>
```

Para obter informações sobre os usuários, use as VIEWS: USER_USERS, ALL_USERS e DBA_USERS. Para saber sobre QUOTAS do usuário, acesse: USER_TS_QUOTAS e DBA_TS_QUOTAS.

6.2 Concedendo acesso ao usuário

Dentro do Oracle, não basta apenas criar um usuário, é preciso conceder acessos (privilégios) para que ele possa manipular os objetos ou até mesmo criá-los no banco de dados. Desta forma, podemos ter, basicamente, dois tipos de privilégios: os de objetos e os de sistemas.

Privilégios de objetos

Os privilégios de objetos são aqueles que permitem o usuário a executar um `SELECT`, `UPDATE`, `DELETE`, `INSERT`, `ALTER`, entre outros.

Já os privilégios de sistema dão ao usuário permissão para administrar o banco de dados. Na lista de privilégios, estão a criação e eliminação de objetos em qualquer `SCHEMA` de usuário.

Segue a lista de alguns privilégios de objetos:

- **SELECT:** permite ao usuário selecionar os dados das tabelas do banco de dados;
- **DELETE:** permite ao usuário excluir registros das tabelas do banco de dados;
- **INSERT:** permite ao usuário inserir registros das tabelas do banco de dados;
- **UPDATE:** permite ao usuário atualizar registros das tabelas do banco de dados;
- **ALTER:** permite ao usuário alterar objetos do banco de dados;
- **EXECUTE:** permite ao usuário executar procedimentos armazenados na base de dados;
- **INDEX:** permite ao usuário criar índices em tabelas os clusters;
- **READ:** permite a leitura de arquivos em um determinado diretório;
- **REFERENCES:** permite ao usuário referenciar uma determinada tabela para a criação de chaves estrangeiras (*FK*);
- **WRITE:** permite a escrita de arquivos em um determinado diretório;
- **ALL [PRIVILEGES]:** contempla todos os privilégios anteriores.

Na sequência, veja exemplos de como dar privilégios de objetos a um usuário. Para isto, utilizamos o comando `GRANT`.

Concedendo as ações de `SELECT`, `UPDATE` e `DELETE`, na tabela `LOCATIONS`, para o usuário `TSQL`:

```
SQL> grant select, update, delete on LOCATIONS to TSQL;

Grant succeeded.

SQL>
```

Concedendo a ação de UPDATE, apenas para os campos POSTAL_CODE E STREET_ADDRESS, na tabela LOCATIONS, para o usuário TSQL:

```
SQL> grant update (POSTAL_CODE, STREET_ADDRESS) on
LOCATIONS to TSQL;

Grant succeeded.

SQL>
```

Concedendo a ação de EXECUTE, da PROCEDURE P_PESQUISA_EMPREGADOS, para o usuário TSQL:

```
SQL> grant execute on P_PESQUISA_EMPREGADOS to TSQL;

Grant succeeded.

SQL>
```

Pare retirar os privilégios dados a um usuário, utilizamos o comando REVOKE. Veja os exemplos a seguir.

Revogando a ação UPDATE, na tabela LOCATIONS, para o usuário TSQL.

```
SQL> revoke update on LOCATIONS from TSQL;

Revoke succeeded.

SQL>
```

Revogando a ação EXECUTE, da PROCEDUREP_PESQUISA_EMPREGADOS, para o usuário TSQL.

```
SQL> revoke execute on P_PESQUISA_EMPREGADOS from TSQL;
```

```
Revoke succeeded.

SQL>
```

> Ao contrário do `GRANT`, o `REVOKE` só pode ser usado com `UPDATE/REFERENCES` a partir da tabela inteira, não sendo possível revogar apenas de uma coluna.

Outro recurso que pode ser usado no momento de executar o comando de `GRANT` é o `WITH GRANT OPTION`. Ele permite ao usuário que está recebendo a concessão o poder de também dar acesso para outros usuários. Vejo o exemplo:

```
SQL> grant select on JOB_HISTORY to TSQL with grant option;

Grant succeeded.

SQL>
```

Neste exemplo, além da concessão de acesso dada ao usuário `TSQL`, também está sendo dada a permissão para que ele possa dar o mesmo tipo de acesso a outros usuários do banco de dados.

6.3 Privilégios de sistema

Este tipo de privilégio permite ao usuário manipular qualquer objeto da base de dados, de qualquer `SCHEMA` (indicando a expressão `ANY`), ou ainda executar comandos relacionados à administração do banco. Segue uma lista com alguns privilégios de sistema:

- **CREATE TABLE:** permite a criação de tabelas;
- **CREATE SYNONYM:** permite a criação de sinônimos;
- **CREATE PUBLIC SYNONYM:** permite a criação de sinônimos públicos;

- **CREATE SEQUENCE:** permite a criação de `SEQUENCES`;
- **CREATE VIEW:** permite a criação de `VIEWS`;
- **CREATE PROCEDURE:** permite a criação de `PROCEDURES`;
- **CREATE TRIGGER:** permite a criação de `TRIGGERS`;
- **SELECT ANY TABLE:** permite a consulta em tabelas de qualquer `SCHEMA`;
- **CREATE ANY INDEX:** permite a criação de índices em qualquer `SCHEMA`;
- **CREATE ANY TABLE:** permite a criação de tabelas em qualquer `SCHEMA`;
- **CREATE ANY TRIGGER:** permite a criação de `TRIGGERS` em qualquer `SCHEMA`;
- **DROP ANY SEQUENCE:** permite a exclusão de `SEQUENCES` em qualquer `SCHEMA`;
- **EXECUTE ANY PROCEDURE:** permite a execução em `PROCEDURES` de qualquer `SCHEMA`;
- **DROP TABLESPACE:** permite a exclusão de qualquer `TABLESPACE`;
- **INSERT ANY TABLE:** permite a inserção em qualquer tabela de qualquer `SCHEMA`;
- **CREATE SESSION:** permite a conexão ao banco de dados;
- **ALTER SESSION:** permite a alteração na sessão aberta no banco de dados;
- **ALTER USER:** permite a alteração do usuário;
- **ALTER TABLESPACE:** permite a alteração de `TABLESPACE`;
- **ALTER ANY TRIGGER:** permite a alteração de `TRIGGERS` em qualquer `SCHEMA`;

- **CREATE ANY DIRECTORY:** permite a criação de `DIRECTORIES` em qualquer `SCHEMA`;
- **DROP ANY DIRECTORY:** permite a exclusão de `DIRECTORIES` em qualquer `SCHEMA`.

Exemplos de concessão:

Concedendo a ação de `ALTER USER` para o usuário `TSQL`:

```
SQL> grant alter user to TSQL;

Grant succeeded.

SQL>
```

Concedendo a ação de `CREATE ANY TABLE` para o usuário `TSQL`:

```
SQL> grant create any table to TSQL;

Grant succeeded.

SQL>
```

Exemplos de revogação:

Revogando a ação de `DROP ANY SEQUENCE` para o usuário `TSQL`:

```
SQL> revoke drop any sequence from TSQL;

Revoke succeeded.

SQL>
```

Revogando a ação de `CREATE ANY DIRECTORY` para o usuário `TSQL`:

```
SQL>  revoke create any directory from TSQL;

Revoke succeeded.

SQL>
```

Assim como em privilégios de objetos, quando concedemos privilégios de sistemas para os usuários também podemos permitir que ele conceda estes privilégios a outros. Fazemos isto utilizando a cláusula `WITH ADMIN OPTION`.

Usuário PUBLIC

Quando estamos trabalhando com um banco de dados grande, onde existem muitos usuários, pode ser dispendioso gerar concessões para cada um. Para isso, existe um usuário chamado `PUBLIC`. Quando concedemos algum privilégio para ele, é como se todos os usuários cadastrados no banco de dados recebessem este privilégio. Contudo, ao contrário dos usuários normais, os quais vimos como manipular, criar, alterar e excluir, este usuário não pode ser utilizado para se conectar no banco de dados. Logo, com ele, não é possível criar, excluir, alterar ou realizar qualquer atividade relacionada aos objetos.

Concedendo a ação de `SELECT`, na tabela `SALGRADE`, para o usuário `PUBLIC`:

```
SQL> grant select on SALGRADE to public;

Grant succeeded.

SQL>
```

6.4 Acesso através de ROLES

Imagine uma `ROLE` como sendo um objeto agrupador de privilégios. Quando criamos uma `ROLE`, podemos incluir em seu teor uma série de privilégios e, melhor, depois de ter feito isto, podemos atribuir esta `ROLE` a um ou mais usuário. Entendeu a ideia? Pois bem, uma `ROLE` nada mais é que um agrupamento de privilégios que, a partir do momento em que é criada e definida, pode ser passada para o usuário como se fosse um privilégio. O usuário, quando recebe como privilégio uma `ROLE`, obtém, automaticamente, todos os privilégios que ela possui. Dentro de uma `ROLE` podemos ter privilégios

de objetos e sistema. Uma ROLE não possui proprietário e pode ser concedida a outra desde que não seja para ela mesma. As ROLES podem ainda ser desativadas e ativadas por usuário. Veja um exemplo:

Criando a ROLE CONTROLA_EMPREGADO:

```
SQL> create role CONTROLA_EMPREGADO;

Role created.

SQL>
```

No passo anterior, apenas criamos a ROLE. Agora temos que definir quais os privilégios ela terá. Vamos permitir a realização de SELECTS na tabela EMP.

```
SQL> grant select on EMP to CONTROLA_EMPREGADO;

Grant succeeded.

SQL>
```

Para o nosso teste, vamos criar um novo usuário chamado TSQL2.

```
SQL> create user TSQL2 identified by TSQL2;

User created.

SQL>
```

Agora, concederemos a ele o privilégio CONTROLA_EMPREGADO, que nada mais é que nossa ROLE criada anteriormente.

```
SQL> grant CONTROLA_EMPREGADO to TSQL2;

Grant succeeded.

SQL>
```

Vamos conectar com o usuário novo:

```
SQL> conn TSQL2/TSQL2@xe2;
ERROR:
ORA-01045: user TSQL2 lacks CREATE SESSION privilege;
           logon denied


Warning: You are no longer connected to ORACLE.
SQL>
```

Houve um erro. O Oracle está informando que o usuário `TSQL2` não possui privilégios para criar uma sessão no banco de dados, ou seja, que ele não pode conectar por falta de privilégios. Note que, realmente, apenas criamos o usuário, mas não dissemos o que ele pode fazer. Para resolver isso, temos que definir alguns privilégios para ele.

Vamos conectar com o usuário `SYSTEM`, que é um dos usuários com mais privilégios dentro do banco de dados Oracle. Através dele, vamos conceder o privilégio de `RESOURCE` e `CONNECT`, que são predefinidos dentro do Oracle. Na verdade, são `ROLES` criadas, contendo uma série de privilégios que permitem aos usuários realizar várias operações dentro do banco de dados, inclusive para conectar-se a ele.

```
SQL> conn system/manager@xe2;
Connected.
SQL>
```

Concedendo alguns privilégios:

```
SQL> grant resource, connect to TSQL2;

Grant succeeded.

SQL>
```

Voltando ao usuário TSQL2, agora conectando no banco de dados com sucesso:

```
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
```

Vamos testar se nossa ROLE está realmente concedida para o usuário TSQL2. Realizaremos um SELECT em cima da tabela EMP do SCHEMA TSQL, no qual nossa ROLE se baseia.

```
SQL> select * from TSQL.EMP where rownum < 10;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81       2975
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000

     COMM     DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10

9 rows selected.

SQL>
```

Perfeito. Nossa ROLE está atribuída. Agora podemos testar o mesmo comando SELECT, mas antes disso vamos revogar o privilégio dado ao usuário. Para isto, conectemos com o usuário TSQL, que também possui privilégios de administrador, e executemos a revogação do privilégio.

```
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>
```

Revogando o privilégio do usuário `TSQL2`:

```
SQL> revoke CONTROLA_EMPREGADO from TSQL2;

Revoke succeeded.

SQL>
```

Estabelecendo conexão com o usuário:

```
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
```

Conforme havíamos previsto, uma vez que retiramos o privilégio do usuário, ele não consegue selecionar os dados da tabela `EMP`.

```
SQL> select * from TSQL.EMP;
select * from TSQL.EMP
                       *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
```

Vamos voltar o privilégio:

```
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>
SQL> grant CONTROLA_EMPREGADO to TSQL2;

Grant succeeded.

SQL>
```

O usuário volta ter privilégios de acesso.

```
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
SQL> select * from TSQL.EMP where rownum < 15;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81       2975
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

      COMM    DEPTNO
 --------- ---------
                  20
       300        30
       500        30
                  20
      1400        30
                  30
                  10
                  20
                  10
         0        30
                  20
                  30
                  20
```

```
                 10

14 rows selected.

SQL>
```

Vamos tentar algo diferente, vamos inserir na tabela EMP:

```
SQL> insert into EMP (empno, ename, job, mgr, hiredate, sal,
                      comm, deptno)
     values (999,'FULANO','VENDEDOR',NULL,'10-JAN-11',3000,10,
             20);
  *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL>
```

Veja que não conseguimos realizar este tipo de operação, pois não foi dado o privilégio de inserção para o usuário e ele também não está contemplado na nossa ROLE. Contudo, isto ainda pode ser feito.

```
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>
SQL> grant insert on EMP to CONTROLA_EMPREGADO;

Grant succeeded.

SQL>
```

Após o privilégio de INSERT ter sido atribuído a ROLE, nosso usuário já pode, agora, realizar inserções na tabela EMP.

```
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
SQL> insert into EMP (empno, ename, job, mgr, hiredate, sal,
                      comm, deptno)
     values (999,'FULANO','VENDEDOR',NULL,'10-JAN-11',3000,10,
```

```
           20);

1 row created.

SQL>
```

Através das ROLES, é possível criar vários cenários para garantir grupos de concessões, como, por exemplo, concessões para gerentes de uma empresa, empregados de uma fábrica ou setor, ou seja, modelar o acesso aos dados através de conjuntos de privilégios, de forma a permitir ou restringir o uso da informação.

Uma ROLE permite o acesso a objetos que estão em outros SCHEMAS, mas não permite que sejam criados outros objetos com base nos objetos que ela deu permissão. Um exemplo disso é a criação de VIEWS com base em tabelas que não estão criadas no SCHEMA do usuário conectado, mesmo ele tendo o privilégio de SELECT e de CREATE VIEW. Veja o exemplo a seguir:

Com o usuário TSQL2 conectado, vamos tentar criar uma VIEW com base na tabela EMP do SCHEMA TSQL (tabela criada neste usuário) à qual foi concedida a permissão de SELECT através da ROLE CONTROLA_EMPREGADO.

```
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
SQL> create view EMP_V as select * from TSQL.EMP;
create view EMP_V as select * from TSQL.EMP
                                         *
ERROR at line 1:
ORA-01031: insufficient privileges


SQL>
```

Note que, nessa execução, não conseguimos executar a criação da VIEW. Um erro informando que os privilégios foram insuficientes nos foi mostrado. Vamos tentar agora conceder um GRANT de CREATE VIEW para o usuário TSQL2. Note que, para isso, temos que conectar com outro usuário que tenha permissão para conceder privilégios. Com esse intuito, utilizamos o usuário TSQL.

```
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>
SQL> grant create view to TSQL;

Grant succeeded.

SQL>
```

Depois do privilégio concedido, vamos tentar, novamente, criar a nossa VIEW com base na tabela de outro SCHEMA.

```
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
SQL> create view EMP_V as select * from TSQL.EMP;
create view EMP_V as select * from TSQL.EMP
                                                *
ERROR at line 1:
ORA-01031: insufficient privileges


SQL>
```

Mesmo com o GRANT de CREATE VIEW, não foi possível criar a nova VIEW. Para testar se o usuário recebeu mesmo o privilégio de criação de VIEW, vamos criar uma tabela chamada EMP2 a partir da tabela EMP do usuário TSQL. Neste caso, é possível realizar a criação do objeto.

Agora você deve estar se perguntando o porquê de dar certo, sendo que estamos criando um objeto também a partir de um objeto concedido. A diferença é fácil de entender. Uma VIEW, embora seja criada no SCHEMA do usuário que está conectado, traz dados de outra tabela que está em outro usuário/schema, ou seja, cada vez que você acessá-la, ela executa o SELECT que a compõe para buscar os dados. Os dados referentes a busca não se encontram no usuário dono da VIEW e, sim, em outro. Neste caso, a VIEW é obrigada a acessar outro SCHEMA.

Como existe esta limitação em ROLES, a execução de sua criação não é permitida. Já a criação de uma tabela, a partir dos dados de outra que está

em outro SCHEMA, faz com que esta tabela que está em outro SCHEMA seja acessada uma única vez, somente no momento da criação da nova, ou seja, quando executamos o CREATE TABLE é feita uma cópia da tabela que está no outro SCHEMA e esta cópia é criada no usuário que executou o comando.

```
SQL> create table EMP2 as select * from TSQL.EMP;

Table created.

SQL>
```

Note que o erro não acontece quando tentamos criar uma VIEW com base na tabela que está criada no mesmo SCHEMA.

```
SQL> create view EMP_V as select * from EMP2;

View created.

SQL>
```

> ROLES não podem ser criadas dentro de outros objetos, por exemplo, dentro de PROCEDURES ou FUNCTIONS.

Atribuindo ROLES por default

Um recurso existente quando estamos trabalhando com ROLES é o de associar uma ROLE por padrão para um usuário. Isso funciona da seguinte forma: se associarmos uma ROLE padrão para um usuário, ele só terá acesso a esta ROLE. Se o usuário não recebeu nenhuma atribuição default, ele poderá usufruir de todas as ROLES que estiverem liberadas para ele. O que deve ser ressaltado é que, primeiramente, uma ROLE deve ser atribuída a um usuário, antes de ser atribuída como seu default. O que deve ser observado, também, é que quando se atribui ROLES por padrão a um usuário, ele perde acesso a outras ROLES que por ventura possa ter, inclusive a de conexão. Para isto, é necessário reatribuí-las, todas como DEFAULT. Veja os tipos de atribuições a seguir:

- **ROLE {<NOME>[, <NOME>]}:** atribui determinada(s) `ROLE`(S) ao usuário;
- **ROLE ALL [EXCEPT ROLE [,ROLE]]:** atribui todas as `ROLES` já concedidas ao usuário, podendo abrir exceções;
- **ROLE NONE:** retira a atribuição das `ROLES` já concedidas ao usuário.

Veja:

```
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>
SQL>
SQL> alter user TSQL2 default role CONTROLA_EMPREGADO;
-- Atribuindo uma ROLE como default.

User altered.

SQL> create role EXCLUI_EMPREGADO; -- Criando a ROLE
EXCLUI_EMPREGADO.

Role created.

SQL> grant delete on EMP to EXCLUI_EMPREGADO; -- Concedendo
ações a ROLE criada.

Grant succeeded.

SQL> grant EXCLUI_EMPREGADO to TSQL2; -- Concedendo a ROLE ao
usuário.

Grant succeeded.

SQL> alter user TSQL2 default role CONTROLA_EMPREGADO,
EXCLUI_EMPREGADO; -- Atribuindo duas ROLES como default.

User altered.

SQL>
```

```
SQL> alter user TSQL2 default role all; -- atribuindo todas as
ROLES como default.

User altered.

SQL> alter user TSQL2 default role all except EXCLUI_EMPREGADO;
-- atribuindo todas as ROLES como default, com exceções.

User altered.

SQL>

SQL> alter user TSQL2 default role none; -- retirando a
atribuição de todas as ROLES default.

User altered.

SQL>
```

Estes exemplos demonstram a criação e atribuição de `ROLES` para o usuário `TSQL2`.

Mesmo que determinadas `ROLES` já tenham sido concedidas ao usuário, podemos fazer com que, por padrão, elas não estejam disponíveis quando o usuário abrir uma sessão no banco de dados. Veja o exemplo:

Vamos conceder dois privilégios para o usuário `TSQL2`:

```
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>
SQL> grant CONTROLA_EMPREGADO to TSQL2;

Grant succeeded.

SQL> grant EXCLUI_EMPREGADO to TSQL2;

Grant succeeded.

SQL>
```

Contudo, vamos configurar o usuário para não receber uma `ROLE` por default quando ele conectar ao banco de dados:

```
SQL> alter user TSQL2 default role none;

User altered.

SQL>
```

Voltemos para o usuário `TSQL2`. Ao tentarmos nos conectar com o usuário `TSQL2`, pode ocorrer um problema por termos definido que ele não teria nenhuma `ROLE` como default. Como já foi dito anteriormente, quando criamos um usuário temos que dar algumas concessões, entre elas, a concessão de conexão ao banco. Esta concessão se torna default pra o usuário, por isso conseguimos conectá-lo ao banco. Quando definimos que não haveria mais `ROLES` default, perdemos esta também. Veja a tentativa a seguir:

```
SQL> conn TSQL2/TSQL2@xe2;
ERROR:
ORA-01045: user TSQL2 lacks CREATE SESSION privilege;
logon denied


Warning: You are no longer connected to ORACLE.
SQL>
SQL> conn TSQL2/TSQL2@xe2;
ERROR:
ORA-01045: user TSQL2 lacks CREATE SESSION privilege;
logon denied


SQL>
SQL> conn TSQL/TSQL@xe2;
Connected.
SQL>

SQL> grant resource, connect to TSQL2;

Grant succeeded.
```

```
SQL> grant create session to TSQL2;

Grant succeeded.

SQL>
SQL> conn TSQL2/TSQL2@xe2;
Connected.
SQL>
SQL>
```

Testando os privilégios concedidos:

```
SQL> select * from TSQL.EMP;
select * from TSQL.EMP
                     *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> delete from TSQL.EMP where DEPTNO = 20;
delete from TSQL.EMP where DEPTNO = 20
                      *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
```

Note que não foi possível selecionar e nem excluir os registros da tabela `EMP`. Isso se deu pelo fato de que, embora o usuário tenha os privilégios concedidos, os mesmos não foram atribuídos a ele. Agora devemos ativar os privilégios para o usuário para que ele possa, enfim, acessar os dados.

Ativando e desativando ROLES

Vimos até aqui que podemos atribuir `ROLES` por padrão aos usuários do banco de dados. Além de atribuirmos uma `ROLE` ou um conjunto de `ROLES` em específico para um usuário, também podemos atribuir todas as `ROLES`

ou nenhuma. No exemplo anterior, embora tenhamos dois privilégios concedidos para o usuário TSQL2, eles não estão disponíveis. Isso ocorreu porque por padrão foi definido que nenhuma ROLE seria colocada como default no momento em que o usuário conectasse, ou seja, elas existem e estão concedidas, mas estão desativadas para este usuário. O que temos que fazer é ativá-las:

```
SQL> set role CONTROLA_EMPREGADO, EXCLUI_EMPREGADO;

Role set.

SQL>
```

Depois de realizada a ativação, as ROLES estarão disponíveis:

```
SQL> select EMPNO, ENAME, JOB, MGR, HIREDATE from TSQL.EMP
where rownum < 5;

    EMPNO ENAME      JOB              MGR HIREDATE
--------- ---------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80
     7499 ALLEN      SALESMAN        7698 20-FEB-81
     7521 WARD       SALESMAN        7698 22-FEB-81
     7566 JONES      MANAGER         7839 02-APR-81

SQL>
```

Outras formas de ativação e desativação de ROLES:

- **ROLE {<NOME> [IDENTIFIED BY PASSWORD] [, <NOME>[IDENTIFIED BY PASSWORD]]}:** atribui determinada(s) ROLE(S) ao usuário, podendo especificar uma senha de acesso;
- **ROLE ALL [EXCEPT ROLE [,ROLE]]:** atribui todas as ROLES já concedidas ao usuário, podendo abrir exceções;
- **ROLE NONE:** retira a atribuição das ROLES já concedidas ao usuário.

Veja. Exemplos de ativação de ROLES em específico:

```
SQL> set role CONTROLA_EMPREGADO;

Role set.

SQL> set role EXCLUI_EMPREGADO identified by senha12345;

Role set.

SQL> set role CONTROLA_EMPREGADO, EXCLUI_EMPREGADO;

Role set.

SQL> set role EXCLUI_EMPREGADO identified by senha12345,
EXCLUI_EMPREGADO identified by senha67890;

Role set.

SQL>
```

Exemplos de ativação de `ROLES` em massa:

```
SQL> set role all;

Role set.

SQL> set role all except EXCLUI_EMPREGADO, CONTROLA_EMPREGADO;

Role set.

SQL>
```

Exemplos de desativação de `ROLES` em massa:

```
SQL> set role none;

Role set.

SQL>
```

Roles predefinidas e informações sobre ROLES

Quando o banco de dados é instalado, algumas `ROLES` são criadas. Elas já vêm com alguns privilégios concedidos. Elas podem ser atribuídas aos usuários para que eles possam conectar ao banco de dados ou até mesmo administrá-lo. Na sequência estão listadas as três mais utilizadas. Existem muitas outras e isto vai depender da versão do banco de dados.

- **CONNECT:** `CREATE SESSION`, `ALTER SESSION`, `CREATE DATABASE LINK`, `CREATE SYNONYM`, `CREATE VIEW`;
- **RESOURCE:** `CREATE TABLE`, `CREATE CLUSTER`, `CREATE SEQUENCE`, `CREATE PROCEDURE`, `CREATE TRIGGER`;
- **DBA:** todos os privilégios de sistema (*with admin option*).

Informações sobre `ROLES`, privilégios de `ROLES` e de usuários podem ser encontradas acessando as seguintes `VIEWS`:

- DBA_ROLES
- DBA_ROLE_PRIVS
- ROLE_TAB_PRIVS
- ROLE_SYS_PRIVS
- ROLE_ROLE_PRIVS

Capítulo 7

Manipulação de tabelas

7.1 Manipulando Tabelas

CREATE TABLE

Este comando é usado para criar uma tabela e definir suas colunas, bem como outras propriedades em um banco de dados.

Veja a criação a tabela EMP:

```
SQL> create table emp
     ( empno                              number(4,0)  not null
      ,ename                              varchar2(10)
      ,job                                varchar2(9)
      ,mgr                                number(4,0)
      ,hiredate                           date
      ,sal                                number(7,2)
```

```
      ,comm                              number(7,2)
      ,deptno                            number(2,0)
     );

Table created.

SQL>
```

Criando a tabela DEPT:

```
SQL> create table dept
     (
       deptno number(2,0)
      ,dname                             varchar2(14)
      ,loc                               varchar2(13)
     );

Table created.

SQL>
```

DEFAULT ESPECIFICATIONS

Na definição da criação da tabela podemos especificar para determinada coluna um valor padrão (DEFAULT). Feito isso, quando fazemos uma inserção de um registro, caso não informemos valor para esta coluna, o Oracle assume o valor que declaramos como DEFAULT. Isso é comumente utilizado em colunas que se referem a datas. Por exemplo, podemos ter um campo na tabela referente à data de registro no sistema de todos os empregados que forem cadastrados. Para isso, definimos um valor default para o campo e, quando inserirmos um registro, este campo receberá este valor padrão. Neste exemplo podemos dizer que o valor DEFAULT é SYSDATE.

Colunas com valores DEFAULT só assumem estes valores caso não sejam informadas no momento de uma inserção, pois, ao informá-las, você terá que definir um valor. Mesmo que seja NULL, o Oracle não acionará o valor DEFAULT, ele assumirá o NULL informado.

Veja a tabela a seguir sendo criada com este tipo de especificação:

```
SQL> create table empregados
       ( empno      number(4,0)  not null
```

```
        ,ename      varchar2(10)
        ,job        varchar2(9)
        ,mgr        number(4,0)
        ,hiredate   date          default sysdate
        ,sal        number(7,2)
        ,comm       number(7,2)   default 0
        ,deptno     number(2,0)
       )
     /

Table created.

SQL> select * from empregados;

no rows selected

SQL>
```

Sem informar a data de admissão (`HIREDATE`), seu uso fica assim:

```
SQL>insert into empregados (empno,ename,job,mgr,sal,comm,deptno)
    values(100,'CEREBRO','PEDREIRO',null,3000,0,10)
     /

1 row created.

SQL>

SQL> select * from empregados;

EMPNO ENAME    JOB       MGR HIREDATE   SAL      COMMDEPTNO
----- -------- --------- --- ---------- ----- ---------- ----------
  100 CEREBRO PEDREIRO       09-APR-11  3000          0         10

SQL>
```

Note que a data de admissão foi preenchida com a data atual do momento em que o comando foi executado.

Se informarmos a data de admissão como nula:

```
SQL> insert into empregados (empno,ename,job,mgr,hiredate,sal,
                                  comm,deptno)
     values(101,'PINK','SERVENTE',100,null,1000,0,10)
     /

1 row created.

SQL>

SQL> select * from empregados;

EMPNO      ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- ---------- ----------
100        CEREBRO    PEDREIRO             09-APR-11       3000
101        PINK       SERVENTE         100                 1000

     COMMDEPTNO
--------- ---------
        0         10
        0         10

SQL>
```

Nesse caso, como a coluna foi informada, embora tenha sido passado o valor nulo, ele foi considerado. Com isto, o valor `DEFAULT` não foi atribuído.

Ao inserir sem o valor de comissão (`COMM`):

```
SQL> insert into empregados (empno,ename,job,mgr,hiredate,sal,
                                deptno)
  2  values(102,'JOCA','SERVENTE',100,sysdate,1000,10)
  3  /

1rowcreated.

SQL>

SQL>select * from empregados;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- ---------- ---------- ---------- ----------
```

```
     100 CEREBRO    PEDREIRO                09-APR-11      3000
     101 PINK       SERVENTE           100                 1000
     102 JOCA       SERVENTE           100  09-APR-11      1000

    COMMDEPTNO
---------- ----------
         0         10
         0         10
         0         10

SQL>
```

Note que a comissão ficou zerada, uma vez que não foi atribuída no comando `INSERT`.

Criando tabela com base em SELECT

No Oracle, é possível criar uma tabela através da seleção de outra tabela.

Veja a seguir a criação da tabela `SALES`, com base na tabela `EMP`, na qual somente empregados vendedores são armazenados.

```
SQL> create table SALES as select * from emp where
     job = 'SALESMAN';

Table created.

SQL>
```

Agora veja a criação da tabela `ANALYST`, também com base na tabela `EMP`, onde somente empregados analistas são armazenados.

```
SQL> create table ANALYST as select * from emp where
     job = 'ANALYST';

Table created.

SQL>
```

Este comando não apenas cria a tabela utilizando a estrutura de uma já existente, como também leva seus dados cadastrados. Além disso, esses dados

não precisam ser confirmados, pois o próprio comando `CREATE` gera um `COMMIT` implicitamente, assim como todo comando `DDL`.

Caso só deseje a estrutura, e não os dados contidos na tabela, use o comando desta forma:

```
SQL> create table CLERK as select * from emp where 1<>1;

Table created.

SQL> desc clerk
 Name                           Null?    Type
 ------------------------------- -------- ----
 EMPNO                          NOT NULL NUMBER(4)
 ENAME                                   VARCHAR2(10)
 JOB                                     VARCHAR2(9)
 MGR                                     NUMBER(4)
 HIREDATE                                DATE
 SAL                                     NUMBER(7,2)
 COMM                                    NUMBER(7,2)
 DEPTNO                                  NUMBER(2)

SQL> select * from clerk;

no rows selected

SQL>
```

Note que a cláusula `WHERE 1<>1`, faz com que nenhum dado seja retornado.

Tabelas temporárias

O Oracle possui um recurso na criação de tabelas que é a tabela temporária. Basicamente, ela funciona da seguinte forma. A princípio, ela age como uma tabela normal: sua criação é feita como qualquer tabela Oracle. A diferença está no armazenamento de dados. Em uma tabela normal, inserimos informações e, se desejarmos que elas permaneçam salvas no banco, nós as efetivamos através do `COMMIT`. No caso das tabelas temporárias, seus dados

ficam armazenados somente enquanto a transação ou sessão no Oracle (por exemplo, no SQL*Plus) estiver aberta. Quando a transação ou sessão é fechada, todos os dados são excluídos e a tabela volta a ficar zerada. Daí vem o conceito de tabela temporária, já que seus dados são temporários numa transação ou sessão.

Temos dois tipos de tabelas temporárias: temporária por transação e temporária por sessão. As tabelas temporárias por transação guardam seus dados enquanto a transação não for encerrada por um `COMMIT` ou `ROLLBACK`. Já as temporárias por sessão guardam seus dados até que a sessão seja encerrada. Vale lembrar que quando conectamos ao banco de dados, através do SQL*Plus, por exemplo, o Oracle abre uma sessão. Quando efetuamos o primeiro comando dentro desta sessão é que o Oracle, implicitamente, abre uma transação. Além disso, estando dentro de uma transação, podemos abrir várias outras transações.

No exemplo a seguir, vamos criar uma tabela chamada `EMP_TMP` com base no `SCRIPT` da tabela `EMP`, que já conhecemos. A única mudança é que vamos criá-la no formato de tabela temporária.

Temporária por sessão

```
SQL> create global temporary table emp_tmp
      ( empno                         number(4,0)  not null
       ,ename                         varchar2(10)
       ,job                           varchar2(9)
       ,mgr                           number(4,0)
       ,hiredate                      date
       ,sal                           number(7,2)
       ,comm                          number(7,2)
       ,deptno                        number(2,0)
      )
     on commit preserve rows;

Table created.

SQL>
```

Vamos inserir dados na tabela a partir da tabela `EMP`.

```
SQL> insert into emp_tmp select * from emp where deptno = 10;

4 rows created.

SQL> select * from emp_tmp;

EMPNO ENAME  JOB        MGR  HIREDATE   SAL   COMM    DEPTNO
----- ------ --------- ----- ---------  ----- ------ --------
 7782 CLARK  MANAGER    7839 09-JUN-81  2450              10
 7839 KING   PRESIDENT       17-NOV-81  5000              10
 7934 MILLER CLERK      7782 23-JAN-82  1300              10
    3 JOCA   MECANICO        21-APR-11  5000      0       10

SQL>
```

Vamos fechar a transação.

```
SQL> commit;

Commit complete.

SQL> select * from emp_tmp;

EMPNO ENAME  JOB        MGR  HIREDATE   SAL   COMM    DEPTNO
----- ------ --------- ----- ---------  ----- ------ --------
 7782 CLARK  MANAGER    7839 09-JUN-81  2450              10
 7839 KING   PRESIDENT       17-NOV-81  5000              10
 7934 MILLER CLERK      7782 23-JAN-82  1300              10
    3 JOCA   MECANICO        21-APR-11  5000      0       10

SQL>
```

Note que, na execução anterior, mesmo fechando a transação, os dados permanecem. Vamos fechar a sessão e verificar o que acontece.

```
SQL> select * from emp_tmp;

EMPNO ENAME  JOB        MGR  HIREDATE   SAL   COMM    DEPTNO
----- ------ --------- ----- ---------  ----- ------ --------
 7782 CLARK  MANAGER    7839 09-JUN-81  2450              10
```

```
 7839 KING   PRESIDENT        17-NOV-81  5000                 10
 7934 MILLER CLERK      7782 23-JAN-82  1300                 10
    3 JOCA   MECANICO         21-APR-11  5000        0        10

SQL> disconnect
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> select * from emp_tmp;

no rows selected

SQL>
```

Temporária por transação

```
SQL> create global temporary table emp_tmp
      ( empno                         number(4,0)  not null
       ,ename                         varchar2(10)
       ,job                           varchar2(9)
       ,mgr                           number(4,0)
       ,hiredate                      date
      ,sal                           number(7,2)
       ,comm                          number(7,2)
       ,deptno                        number(2,0)
  );

Table created.

SQL>
```

Igualmente ao que fizemos anteriormente, vamos inserir dados na tabela a partir da tabela `EMP`.

```
SQL> insert into emp_tmp select * from emp where deptno = 10;

4 rows created.

SQL> select * from emp_tmp;
```

```
    EMPNO ENAME      JOB              MGR HIREDATE          SAL
--------- ---------- --------- --------- --------- ---------
     7782 CLARK      MANAGER         7839 09-JUN-81      2450
     7839 KING       PRESIDENT            17-NOV-81      5000
     7934 MILLER     CLERK           7782 23-JAN-82      1300
        3 JOCA       MECANICO             21-APR-11      5000

     COMM    DEPTNO
--------- ---------
                 10
                 10
                 10
        0        10

SQL>
```

Vamos fechar a transação.

```
SQL> select * from emp_tmp;

    EMPNO ENAME      JOB              MGR HIREDATE          SAL
--------- ---------- --------- --------- --------- ---------
     7782 CLARK      MANAGER         7839 09-JUN-81      2450
     7839 KING       PRESIDENT            17-NOV-81      5000
     7934 MILLER     CLERK           7782 23-JAN-82      1300
        3 JOCA       MECANICO             21-APR-11      5000

     COMM    DEPTNO
--------- ---------

                 10
                 10
                 10
        0        10



SQL> commit;

Commit complete.
```

```
SQL> select * from emp_tmp;

no rows selected

SQL>
```

Note que agora o resultado foi outro. Como finalizamos a transação, os dados foram excluídos.

Algumas características das tabelas temporárias:

- Dados de tabelas temporárias só podem ser vistos na sessão que os criou;
- Podemos criar `TRIGGERS` e `VIEWS` para tabelas temporárias assim como para as tabelas comuns;
- O uso de `TRUNCATE`, `ROLLBACK` e `SAVEPOINT` é permitido;
- A definição de uma tabela temporária fica acessível para todas as sessões;
- Transações filhas podem acessar dados de tabelas temporárias da transação pai. Entretanto, se a transação pai modificou algum dado da tabela, a transação filha não poderá usar a tabela. O mesmo acontece se a transação filha estiver utilizando a tabela. Neste caso, nenhuma outra transação, inclusive a transação pai, poderá utilizá-la até a transação ser finalizada. Contudo, feito isso, os dados inseridos na transação filha são excluídos;
- É permitido criar tabelas temporárias em tempo de execução;
- Podemos criar índices para tabelas temporárias, mas eles serão temporários;
- É permitida a criação de concessões (`GRANTS`) para as tabelas temporárias, bem como sinônimos;
- Podemos utilizar tabelas temporárias em `PROCEDURES`, `FUNCTIONS` e `PACKAGES`.

7.2 ALTER TABLE

Através deste comando, você pode alterar a estrutura de uma tabela, incluindo novas colunas, alterando o tipo de dado, alterando a obrigatoriedade e criando novas CONSTRAINTS, como chaves primárias e estrangeiras.

Adicionando a coluna DT_NASCIMENTO, do tipo DATE, na tabela EMP:

```
SQL> alter table EMP add DT_NASCIMENTO date;

Table altered.

SQL> desc EMP
Name                            Null?    Type
------------------------------- -------- ----
EMPNO                           NOT NULL NUMBER(4)
ENAME                                    VARCHAR2(10)
JOB                                      VARCHAR2(9)
MGR                                      NUMBER(4)
HIREDATE                                 DATE
SAL                                      NUMBER(7,2)
COMM                                     NUMBER(7,2)
DEPTNO                                   NUMBER(2)
DT_NASCIMENTO                            DATE

SQL>
```

Excluindo a coluna DT_NASCIMENTO da tabela EMP:

```
SQL> alter table EMP drop column DT_NASCIMENTO;

Table altered.

SQL> desc EMP;
Name                            Null?    Type
------------------------------- -------- ----
EMPNO                           NOT NULL NUMBER(4)
ENAME                                    VARCHAR2(10)
JOB                                      VARCHAR2(9)
MGR                                      NUMBER(4)
```

```
HIREDATE                                DATE
SAL                                     NUMBER(7,2)
COMM                                    NUMBER(7,2)
DEPTNO                                  NUMBER(2)

SQL>
```

Modificando a coluna `ENAME` para `VARCHAR2(240) NOT NULL`, antes `NULL`:

```
SQL> alter table EMP modify ENAME varchar2(240) not null;

Table altered.

SQL> DESC EMP;
 Name                             Null?    Type
 -------------------------------- -------- ----
 EMPNO                            NOT NULL NUMBER(4)
 ENAME                            NOT NULL VARCHAR2(240)
 JOB                                       VARCHAR2(9)
 MGR                                       NUMBER(4)
 HIREDATE                                  DATE
 SAL                                       NUMBER(7,2)
 COMM                                      NUMBER(7,2)
 DEPTNO                                    NUMBER(2)

SQL>
```

Adicionando a coluna `DS_OBSERVACAO`, do tipo `VARCHAR2(2000)`, na tabela `EMP`.

```
SQL> alter table EMP add (ds_observacao varchar2(2000));

Table altered.

SQL> DESC EMP;
 Name                             Null?    Type
 -------------------------------- -------- ----
 EMPNO                            NOT NULL NUMBER(4)
 ENAME                            NOT NULL VARCHAR2(240)
```

```
 JOB                                       VARCHAR2(9)
 MGR                                       NUMBER(4)
 HIREDATE                                  DATE
 SAL                                       NUMBER(7,2)
 COMM                                      NUMBER(7,2)
 DEPTNO                                    NUMBER(2)
 DS_OBSERVACAO                             VARCHAR2(2000)

SQL>
```

Modificando a coluna `DS_OBSERVACAO` para `VARCHAR2(4000)`, antes `VARCHAR2(2000)`:

```
SQL> alter table EMP modify ds_observacao varchar2(4000);

Table altered.

SQL> DESC EMP;
 Name                               Null?    Type
 ------------------------------- -------- ----
 EMPNO                              NOT NULL NUMBER(4)
 ENAME                              NOT NULL VARCHAR2(240)
 JOB                                         VARCHAR2(9)
 MGR                                         NUMBER(4)
 HIREDATE                                    DATE
 SAL                                         NUMBER(7,2)
 COMM                                        NUMBER(7,2)
 DEPTNO                                      NUMBER(2)
 DS_OBSERVACAO                               VARCHAR2(4000)

SQL>
```

Renomeando uma coluna da tabela:

```
SQL> alter table EMP rename column ename to nome;

Table altered.

SQL>
```

Através do comando ALTER TABLE, também é possível colocar a tabela em modo de consulta, impedindo que alterações de DDL ou DML sejam realizadas. Consequentemente, a sua volta ao modo de gravação também é realizada por este comando.

```
SQL> alter table emp read only; -- configura para somente
leitura.

Table altered.

SQL>
SQL> alter table emp read write; -- configura para leitura e
gravação.

Table altered.

SQL>
```

7.3 DROP TABLE

Assim como você pode criar uma tabela, você também pode excluí-la, desde que você tenha poderes para isto.

Excluindo a tabela EMP:

```
SQL> drop table EMP;

Table dropped.

SQL>
```

Excluindo a tabela DEPT:

```
SQL> drop table DEPT;

Table dropped.

SQL>
```

7.4 TRUNCATE TABLE

O comando TRUNCATE elimina todas as linhas da tabela. Diferente do DELETE, a ação deste comando não permite ROLLBACK, ou seja, uma vez executado, não há como voltar à situação anterior, pois ele faz um COMMIT implicitamente. Por isso, ele também pode ser considerado um comando DDL. Entretanto, ele não elimina a estrutura da tabela ou de índices, somente os dados.

Se existirem privilégios, TRIGGERS ou CONSTRAINTS, eles continuarão intactos. No caso das TRIGGERS, elas não são disparadas quando o comando é executado. Um cuidado que se deve ter é desabilitar as chaves estrangeiras (FKs) antes de executar este comando, a menos que seja uma chave de autorrelacionamento (SELF-JOIN).

Truncando a table EMP:

```
SQL> truncate table EMP;

Table truncated.

SQL>
```

7.5 Ordenando dados

ORDER BY

Este comando especifica a ordem em que as linhas selecionadas serão exibidas. Estas colunas podem ou não estar dispostas no SELECT. Para definir a ordenação, deve-se colocar a cláusula ORDER BY e, logo após, o nome da coluna ou sua posição na cláusula SELECT, caso ela esteja informada. Além disso, você pode definir qual tipo de ordenação:

- ASC – ordenação ascendente (quando não se define qual tipo de ordenação, esta será a padrão)
- DESC – ordenação descendente

Ordenando por um nome de coluna. Ordenação do tipo ASC, pois o tipo não foi informado:

```
SQL> select ename
     from emp
     order by ename;

ENAME
----------
ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD

14 rows selected.

SQL>
```

A ordenação também pode ser feita informando a posição da coluna, na sequência em que ela aparece no `SELECT`. Neste exemplo, a ordenação está sendo realizada pela coluna 2 (`FIRST_NAME`).

```
SQL> select employee_id
           ,first_name
           ,email
     from   employees
     where rownum < 15
     order by 2;

EMPLOYEE_ID FIRST_NAME           EMAIL
----------- -------------------- -------------------------
        103 Alexander            AHUNOLD
```

```
       104 Bruce                    BERNST
       109 Daniel                   DFAVIET
       105 David                    DAUSTIN
       107 Diana                    DLORENTZ
       111 Ismael                   ISCIARRA
       110 John                     JCHEN
       112 Jose Manuel              JMURMAN
       102 Lex                      LDEHAAN
       113 Luis                     LPOPP
       108 Nancy                    NGREENBE
       101 Neena                    NKOCHHAR
       100 Steven                   SKING
       106 Valli                    VPATABAL

14 rows selected.

SQL>
```

Ordenação pela coluna SALARY.

```
SQL> select first_name
          ,email
          ,salary
     from employees
     where salary between 4000 and 5000
     order by salary;

FIRST_NAME           EMAIL                         SALARY
-------------------- ------------------------- ----------
Sarah                SBELL                           4000
Alexis               ABULL                           4100
Diana                DLORENTZ                        4200
Nandita              NSARCHAN                        4200
Jennifer             JWHALEN                         4400
Valli                VPATABAL                        4800
David                DAUSTIN                         4800

7 rows selected.

SQL>
```

Ordenando pela coluna FIRST_NAME, agora informando o tipo de ordenação ASC (ascendente):

```
SQL> select first_name
           ,email
           ,salary
     from employees
     where job_id = 'IT_PROG'
     order by first_name asc;

FIRST_NAME           EMAIL                      SALARY
-------------------- ------------------------- ---------
Alexander            AHUNOLD                        9000
Bruce                BERNST                         6000
David                DAUSTIN                        4800
Diana                DLORENTZ                       4200
Valli                VPATABAL                       4800

SQL>
```

Além das ordenações vistas anteriormente, podemos escolher o tipo de ordenação por coluna:

```
SQL> select job
             ,sal
     from emp
     where deptno = 20
     order by job asc, sal desc;

JOB             SAL
--------- ---------
ANALYST        3000
ANALYST        3000
CLERK          1100
CLERK           800
MANAGER        2975

SQL>
```

Neste exemplo, estamos ordenando da seguinte forma. Primeiramente, os registros são ordenados de forma ascendente pela coluna JOB. Logo em seguida, é aplicada a segunda ordenação pelo maior salário (coluna SAL), considerando a primeira ordenação já aplicada, ou seja, dentro de um conjunto de linhas de um mesmo cargo, será realizada a ordenação por salário.

Também pode-se utilizar o comando SELECT juntamente com o ORDER BY para ordenar informações com base em colunas de outras tabelas.

```
SQL> select empno
           ,ename
           ,deptno
     from emp e
     where rownum < 11
     order by (select dname
               from dept d
               where d.deptno = e.deptno);

    EMPNO ENAME          DEPTNO
--------- ---------- ---------
     7839 KING               10
     7782 CLARK              10
     7788 SCOTT              20
     7566 JONES              20
     7369 SMITH              20
     7698 BLAKE              30
     7844 TURNER             30
     7521 WARD               30
     7499 ALLEN              30
     7654 MARTIN             30

10 rows selected.

SQL>
```

Neste exemplo, a ordenação está sendo realizada por valores vindos de uma *subquery*. No caso, estes valores são os nomes dos departamentos selecionados.

7.6 Trazendo dados distintos

DISTINCT

Este comando é usado no caso de uma consulta apresentar mais de um registro com valores idênticos. Usando-o, não será permitida reincidência de registros já exibidos. Veja o exemplo:

```
SQL> select deptno
     from emp
     order by deptno asc;

   DEPTNO
---------
       10
       10
       10
       20
       20
       20
...
       30
       30
       30
       30

14 rows selected.

SQL>

SQL> select distinct deptno
     from emp
     order by deptno asc;

   DEPTNO
---------
       10
       20
       30
```

```
SQL>
```

Neste exemplo, um `DISTINCT` pela coluna `DEPTNO` faz com que seja mostrada apenas uma ocorrência de cada código de departamento.

Em vez de usar o comando `DISTINCT` também podemos usar o `UNIQUE`. Ambos são semelhantes.

Sabemos que o `DISTINCT` é utilizado para restringir linhas duplicadas. Também sabemos que, em um `SELECT` que esteja gerando linhas duplicadas, por padrão, o Oracle traz todas. Entretanto, também existe o inverso do `DISTINCT`, chamado `ALL`. Embora quase não o vemos, ele pode ser usado; entretanto, não faz realmente diferença, pois seu resultado seria igual a não informá-lo.

7.7 Relacionamento entre tabelas

É muito comum quando se está selecionando determinadas informações surgir a necessidade de acessar outras tabelas com um mesmo comando `SELECT`, para que seja possível reunir seus dados em um único resultado. Através da cláusula `WHERE`, conseguimos fazer a ligação entre tabelas e, consequentemente, criar um vínculo entre as informações. A esta ligação o Oracle intitula de "*Join*", ou seja, relacionamento. Os tipos de relacionamento existentes são: `INNER JOIN` (também chamado do `SIMPLE JOIN`), `OUTER JOIN` e `SELF JOIN`. Veja as características de cada um adiante.

7.8 INNER JOIN

Para que esta cláusula seja verdadeira, é necessário que as duas colunas que estão sendo comparadas possuam o mesmo valor, ou seja, é preciso que tenha equivalência nas duas tabelas. Exemplo:

```
SQL> select ename, dname
  2  from emp e, dept d
  3  where e.deptno = d.deptno;

ENAME      DNAME
```

```
---------- --------------
SMITH      RESEARCH
ALLEN      SALES
WARD       SALES
JONES      RESEARCH
MARTIN     SALES
BLAKE      SALES
CLARK      ACCOUNTING
SCOTT      RESEARCH
KING       ACCOUNTING
TURNER     SALES
ADAMS      RESEARCH
JAMES      SALES
FORD       RESEARCH
MILLER     ACCOUNTING

14 rows selected.

SQL>
```

Note que, neste exemplo, existe um relacionamento entre as tabelas `EMP` e `DEPT`, através da coluna `DEPTNO`. Este relacionamento é que faz as duas tabelas conversarem e retornarem as informações correspondentes. Se olharmos separadamente cada tabela, podemos observar que nem todos os departamentos apareceram no resultado do `SELECT`. Observe a tabela `DEPT`:

```
SQL> select * from dept order by deptno;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON

SQL>
```

Agora olhe a tabela EMP:

```
SQL> select ename, deptno
  from emp
  order by 2;

ENAME          DEPTNO
---------- ---------
CLARK              10
KING               10
MILLER             10
JONES              20
FORD               20
ADAMS              20
SMITH              20
SCOTT              20
WARD               30
TURNER             30
ALLEN              30
JAMES              30
BLAKE              30
MARTIN             30

14 rows selected.

SQL>
```

Se compararmos a coluna `DEPTNO` da tabela `EMP` com a da tabela `DEPT` vamos perceber que o departamento `OPERATIONS` (código 40) não aparece para nenhum empregado. Logo, não há uma relação entre este departamento com algum empregado cadastrado. Sendo assim, eles não aparecem no `SELECT` onde listamos empregados com seus respectivos departamentos. Este é o tipo mais comum de relacionamento entre tabelas usados no Oracle: duas ou mais tabelas se relacionando para formar um único resultado em um comando SQL.

O Oracle, em versões anteriores à 9i, tinha uma forma peculiar de trabalhar com relacionamentos entre tabelas. Ele possuía uma sintaxe própria, ou seja, não utilizava a sintaxe do padrão ANSI (SQL92) para estes relacionamentos. Contudo, o padrão ANSI (*American National Standards Institute*) ficou forte nas versões mais recentes do banco e está se tornando o novo pa-

drão recomendado pela própria Oracle, por vários motivos, inclusive com a proposta de ganhos de desempenho através do seu uso.

Desta forma, para cada tipo de relacionamento explicado aqui, colocarei também a forma de escrevê-lo no padrão ANSI. Assim, você terá a oportunidade de conhecer o padrão Oracle, utilizado em versões anteriores do banco de dados, e também nas versões novas pois a Oracle mantém o seu padrão por questões históricas (pelo menos por enquanto) e você também verá como escrever relacionamentos entre tabelas utilizando o padrão ANSI, o novo padrão suportado pela Oracle.

Padrão ANSI SQL92 – INNER JOIN

Conforme a documentação da Oracle para as versões mais recentes do banco, por exemplo, a versão 11G, existe ainda outra forma de classificar e utilizar os relacionamentos `INNER JOIN` entre as tabelas. São os chamados `JOINSNATURAL`. Dentro deste conceito, são utilizadas as cláusulas `NATURAL JOIN`, `USING` e `ON` para montar outras formas de relacionamento. Só precisamos ressaltar o que já conversamos sobre as versões anteriores à versão Oracle9i: a sintaxe dos `JOINS` era diferente dos padrões ANSI. Desta forma, este novo padrão, que está em conformidade com o SQL:1999, não funciona em versões anteriores. Conforme documentação da própria Oracle, para versõesanteriores à 9i, este novo padrão não oferecia benefícios de desempenho para a sintaxe de `JOIN`.Talvez, esse seria o motivo pelo qual a Oracle não tenha adotado este padrão desde o início.

Uma das coisas que você vai notar é que no padrão ANSI existe mais de uma forma de escrever um relacionamento do tipo `INNER JOIN`. Mas não se preocupe, analisando estas formas você também vai perceber que o intuito foi simplificar a escrita e tornar mais clara a leitura do comando SQL. Os conceitos são os mesmos, inclusive para os outros tipos de realcionamento, desde o `OUTER JOIN` ou até mesmo o `SELF JOIN`. A seguir, alguns exemplos e regras utilizando estas cláusulas.

Cláusula NATURAL JOIN

Para caracterizarmos um relacionamento do `NATURAL JOIN`, precisamos seguir a sua premissa básica: a possibilidade de unir duas tabelas com

base no mesmo nome de coluna. Veja algumas regras com relação a este tipo de relacionamento:

- A cláusula NATURAL JOIN tem como base todas as colunas das duas tabelas que têm o mesmo nome;
- Ela seleciona as linhas a partir de duas tabelas que têm valores iguais em todas as colunas correspondentes;
- Se as colunas que têm os mesmos nomes tiverem tipos de dados diferentes, será retornado um erro.

Veja um exemplo:

```
SQL> select dname, ename, sal
     from   dept
     natural join emp
```

Como pode ser visto neste exemplo, estamos selecionando os nomes dos funcionários, nomes dos empregados e seus respectivos salários. Note que o relacionamento entre as duas tabelas foi realizadoatravés de um método INNER JOIN, utilizando a cláusula NATURAL JOIN para relacioná-las. Neste caso, não é necessário relacionar as colunas de forma explícita, pois as duas tabelas estão sendo relacionadas por este tipo de cláusula, a NATURAL JOIN. Portanto, todo otrabalho fica a cargo do motor do banco de dados. Ele irá consultar adefinição das tabelas utilizadas no comando SQL à procura das colunas que combinam conforme as regras definidas para este tipo de relacionamento, colunas com o mesmo nome, mesmo tipo de dado e tamanho.

7.9 Cláusula USING

Enquanto no uso das NATURAL JOINS você não necessita especificar as colunas, pois neste caso é preciso apenas existirem colunas com os mesmos nomes e tipo de dados entre as tabelas, nos relacionamentos utilizando a cláusula USING é necessário especificá-las. Na verdade, é para exatamente este fim que esta cláusula serve. Ela é utilizada para quando você necessitar especificar, de formaexplícita, os nomes das colunas. Veja a seguir, em quais casos:

- Se várias colunas tiverem os mesmos nomes, mas tipos de dados que não correspondem, usecláusula `USING` para especificar as colunas para a `EQUIJOIN` (`EQUIJOINS` também são chamadas de `%SIMPLE JOINS` ou `INNER JOINS`);
- Use a cláusula `USING` para corresponder somente uma coluna quando houver mais de uma coluna correspondente;
- As cláusulas `NATURAL JOIN` e `USING` são mutuamente exclusivas;
- Frequentemente, este tipo de join envolve complementos de chave primária e estrangeira.

Veja um exemplo on é mostrada aunião da tabela de empregados e departamentos, com intuito de recuperar a localização (`LOCATION_ID`) de cada empregado.

```
SQL> select employee_id, last_name
   location_id, department_id
   from employees JOIN departments
   USING (department_id);

EMPLOYEE_ID LOCATION_ID DEPARTMENT_ID
----------- ------------------------- -------------
        100 King                          90
        101 Kochhar                          90
        102 De Haan                          90
        103 Hunold                          60
        104 Ernst                          60
        105 Austin                          60
        106 Pataballa                          60
        107 Lorentz                          60
        108 Greenberg                          100
        ...                                    000
        205 Higgins                          110
        206 Gietz                          110

106 rows selected.

SQL>
```

Regras para o uso de apelidos de tabela com o uso da cláusula USING

- Não qualifique (comnome de tabela ou apelido) uma coluna que é usada na cláusula
- Se a mesma coluna for usada em qualquer outra parte da instrução SQL, não lhe dê um apelido, caso contrário, a execução retornará um erro.
- Colunas que são comuns nas duas tabelas, mas não são usadas na cláusula USING devem ser prefixadas com um apelido de tabela, caso contrário, você receberá a seguinte mensagem de erro "column ambiguously defined."

Veja um exemplo:

```
SQL> select l.city, d.department_name
    from locations l JOIN departments d
    USING (location_id)
    where d.location_id = 1400;
   where d.location_id = 1400
 *

ERROR at line 4:
ORA-25154: column part of USING clause cannot have qualifier

SQL>
```

Note, neste exemplo, que ao colocarmos um apelida na coluna LOCATION_ID, que se trata de uma coluna utilizada na cláusula USING, um erro foi gerado. Veja agora, como deveria ser este comando:

```
SQL> select l.city, d.department_name
    from locations l JOIN departments d
    USING (location_id)
    where location_id = 1400
    /

CITY DEPARTMENT_NAME
```

```
------------------------------ ------------------------------
Southlake IT

SQL>
```

Cláusula ON

A cláusula ON possibilita você a tornar explícita todas as condições de relacionamentos entre tabelas. Não só aquelas exclusivas, aquelas que possuem ligações de chaves primárias com chaves estrangeira. Desta forma, para casos que não for possível utilizar as cláusulas `NATURAL JOIN` ou `USING`, utilize a cláusula `ON`. A seguir, algumas características:

- A condição de `JOIN` para a `JOIN NATURAL` é basicamente uma equijoin de todas as colunas com o mesmo nome. Ou seja, o era implícito antes, pode agora de colocado de forma explícita através do uso da cláusula `ON`.
- Use a cláusula `ON` para especificar condições arbitrárias, ou seja, condições não lógicas, que fugiriam as regras de um `NATURAL JOIN`, ou especificar colunas a serem unidas. Por exemplo, colunas com nomes diferentes.
- A condição de `JOIN` é separada de outras condições de pesquisa ou filtros na cláusula `WHERE`.
- A cláusula `ON` facilita a compreensão do código, pois torna tudo mais explícito e de forma organizada.
- Use apelidos de tabelas para qualificar as colunas utilizadas na cláusula `ON`, quando os nomes das colunas forem iguais.

Veja um exemplo:

```
SQL> select e.employee_id, e.last_name, e.department_id,
     d.department_id, d.location_id
    from employees e JOIN departments d
    ON (e.department_id = d.department_id)
```

```
    /

EMPLOYEE_ID LAST_NAME DEPARTMENT_ID DEPARTMENT_ID LOCATION_ID
----------- ------------ ------------- ------------- -----------
        100 King                    90            90        1700
        101 Kochhar                 90            90        1700
        102 De Haan                 90            90        1700
        103 Hunold                  60            60        1400
        104 Ernst                   60            60        1400
        105 Austin                  60            60        1400
        106 Pataballa               60            60        1400
...
        201 Hartstein               20            20        1800
        202 Fay                     20            20        1800
        203 Mavris                  40            40        2400
        204 Baer                    70            70        2700
        205 Higgins                110           110        1700
        206 Gietz                  110           110        1700

106 rows selected.

SQL>
```

Veja que neste exemplo, as tabelas foram unidas pela coluna DEPARTMENT_ID na cláusula `ON`, sendo que as colunas foram rotuladas com os respectivos apelidos, conforme definido nas tabelas. Vejamos agora, outros dois exemplos, um mostrando como trabalhar com várias tabelas utilizando a cláusula `ON` e, também, outro exemplo mostrando como inserir condições adicionais em uma JOIN.

Trabalhando com mais de duas tabelas usando a cláusula ON:

```
SQL> select employee_id, city, department_name
      from employees e
     JOIN departments d
     ON d.department_id = e.department_id
     JOIN locations l
     ON d.location_id = l.location_id;

EMPLOYEE_ID CITY                           DEPARTMENT_NAME
```

```
----------- ------------------------------ ---------------------
        100 Seattle                        Executive
        101 Seattle                        Executive
        102 Seattle                        Executive
        103 Southlake                      IT
        104 Southlake                      IT
        105 Southlake                      IT
...
        205 Seattle                        Accounting
        206 Seattle                        Accounting

106 rows selected.

SQL>
```

Inserindocondições adicionais em uma `JOIN`, utilizando as cláusulas `AND` e `WHERE`

```
SQL> select e.employee_id, e.last_name, e.department_id,
   d.department_id, d.location_id
   from employees e JOIN departments d
   on (e.department_id = d.department_id)
   and e.manager_id = 149;

EMPLOYEE_ID LAST_NAME    DEPARTMENT_ID DEPARTMENT_ID LOCATION_ID
----------- ------------ ------------- ------------- -----------
        179 Johnson                 80            80        2500
        177 Livingston              80            80        2500
        176 Taylor                  80            80        2500
        175 Hutton                  80            80        2500
        174 Abel                    80            80        2500

SQL>

SQL> select e.employee_id, e.last_name, e.department_id,
   d.department_id, d.location_id
   from employees e JOIN departments d
   on (e.department_id = d.department_id)
   where e.manager_id = 149;
```

```
EMPLOYEE_ID LAST_NAME    DEPARTMENT_ID DEPARTMENT_ID LOCATION_ID
----------- ------------ ------------- ------------- -----------
        179 Johnson                 80            80        2500
        177 Livingston              80            80        2500
        176 Taylor                  80            80        2500
        175 Hutton                  80            80        2500
        174 Abel                    80            80        2500

SQL>
```

No-Equijoins

Relacionamentos entre tabelas que não são constituídos por um operador de igualdade são chamados de `No-Equijoins`. Geralmente, estes relacionamentos envolvem tabelas que possuem uma ligação, contudo, não uma ligação direta através de uma equijoin, mas sim, um relacionamento com base em um grupo ou range de valores. Veja o exemplo a seguir:

```
SQL> select e.last_name, e.salary, j.grade_level
   from employees e JOIN job_grades j
   on e.salary
   between j.lowest_sal and j.highest_sal;

LAST_NAME SALARY GRA
------------------------- ---------- ---
Olson                         3074.61 B
Philtanker                    3221.02 B
Markle                        3221.02 B
Landry                        3513.84 B
Gee                           3513.84 B
Marlow                        3660.25 B
Vargas                        3660.25 B
...
Hartstein                     19033.3 E
Partners                      19765.35 E
Russell                       20497.4 E
De Haan                       24889.7 E
Kochhar                       24889.7 E
King                          35138.4 F
```

```
107 rows selected.

SQL>
```

OUTER JOIN

O uso desta cláusula se dá quando mesmo que não exista a equivalência entre as tabelas, os dados sejam preservados e retornados. Exemplo:

```
SQL> select ename, dname
     from emp e, dept d
     where e.deptno(+) = d.deptno;

ENAME      DNAME
---------- --------------
SMITH      RESEARCH
ALLEN      SALES
WARD       SALES
JONES      RESEARCH
MARTIN     SALES
BLAKE      SALES
CLARK      ACCOUNTING
SCOTT      RESEARCH
KING       ACCOUNTING
TURNER     SALES
ADAMS      RESEARCH
JAMES      SALES
FORD       RESEARCH
MILLER     ACCOUNTING
OPERATIONS

15 rows selected.

SQL>
```

Assim como foi visto no exemplo com `INNER JOIN`, aqui também vemos um relacionamento entre as tabelas EMP e DEPT, através da coluna DEPTNO. Contudo, também percebemos a expressão (+) que indica ao co-

mando SQL que mesmo que não haja determinados códigos de departamento na tabela EMP, os mesmos devem ser mostrados. Esta expressão sempre é colocada junto da coluna onde os valores podem não existir. Neste caso, podemos ter departamentos que não possuem empregados cadastrados. Logo a expressão é colocada ao lado da coluna DEPTNO da tabela EMP. Se formos olhar separadamente cada tabela, podemos observar que nem todos os departamentos existem na tabela EMP. Observe:

```
SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON

SQL>
```

Agora olhe a tabela EMP:

```
SQL> select ename, deptno
     from emp;

ENAME          DEPTNO
---------- ---------
SMITH              20
ALLEN              30
WARD               30
JONES              20
MARTIN             30
BLAKE              30
CLARK              10
SCOTT              20
KING               10
TURNER             30
ADAMS              20
JAMES              30
FORD               20
```

```
MILLER          10

14 rows selected.

SQL>
```

Se compararmos novamente a coluna DEPTNO da tabela EMP com a da tabela DEPT vamos perceber que o departamento OPERATIONS (código 40) não aparece para nenhum empregado. Entretanto, este departamento aparece em nosso resultado, na primeira query pelo fato de o nosso relacionamento ser do tipo OUTER JOIN.

O relacionamento do tipo OUTER JOIN ainda pode ser classificado como LEFT ou RIGHT. Contudo, no padrão Oracle, a diferenciação de um e de outro, LEFT OUTER JOIN ou RIGHT OUTER JOIN, é mais simbólica, pois o que manda é onde o sinal (+) está posicionado. Neste caso, o lado da coluna onde estiver o sinal (+) é o lado onde pode não existir a equivalência. A seguir, veremos o padrão ANSI, onde o conceito de LEFT OUTER JOIN e RIGHT OUTER JOIN se faz mais claro no momento da interpretação de comando SQL.

Padrão ANSI SQL92 – OUTER JOIN

A implentação conceitual do `OUTER JOIN` para o padrão ANSI segue a mesma linha do padrão Oracle. O que muda é a forma como escrevemos o comando e indentificamos o tipo do relacionamento. Só ressaltando, enquanto um relacionamento do tipo `INNER JOIN` objetiva a existência de uma equivalência total entre os lados comparados, o relacionamento do tipo `OUTER JOIN`, prevê situações onde possa não existir esta equivalência em todos os casos.

O cenário mais comum no uso de `OUTER JOINS`, eu diria que 99% dos casos, são relacionamentos de Joins INNER sendo comparadas com Joins OUTER. Desta forma, a união entre duas tabelas que retorna os resultados de uma junção `INNER JOIN` e de linhas sem equivalencia da tabela esquerda é chamada de `LEFT OUTER JOIN`. Logo, o retorno de uma junção `INNER JOIN` com linhas sem correspondência da tabela direta é chamada de `RIGHT OUTER JOIN`. No entanto, ainda existe outro cenário, que é quando existe

uma união entre duas tabelas que retornam os resultados de uma `INNER JOIN` e os resultados de uma `LEFT OUTER JOIN` e `RIGHT OUTER JOIN`. Este tipo de relacionamento `OUTER JOIN` A Oracle chama de `FULL OUTER JOIN`. A seguir, veremos exemplos dos três tipos de `OUTER JOINS` existente: `LEFT OUTER`, `RIGTH OUTER` e `FULL OUTER`.

Exemplo de LEFT OUTER JOIN:

```
SQL> select ename, dept.deptno, loc
     from dept left outer join emp
     on emp.deptno = dept.deptno
     /

ENAME          DEPTNO LOC
---------- --------- -------------
SMITH              20 DALLAS
ALLEN              30 CHICAGO
WARD               30 CHICAGO
JONES              20 DALLAS
MARTIN             30 CHICAGO
BLAKE              30 CHICAGO
CLARK              10 NEW YORK
SCOTT              20 DALLAS
KING               10 NEW YORK
TURNER             30 CHICAGO
ADAMS              20 DALLAS
JAMES              30 CHICAGO
FORD               20 DALLAS
MILLER             10 NEW YORK
EDUARDO            20 DALLAS
                   50 BRASIL
                   40 BOSTON
                   60 BRASIL

18 rows selected.

SQL>
```

Neste exemplo, estamos recuperando todos os empregaos com seus respectivos departamentos e localizações. Note que os departamentos onde não

há empregados alocados, também são retornados pelo comando SELECT. Se analisarmos o comando, estamos selecionando os dados das tabelas DEPT e EMP, relacionando-as através de um relacionamento do tipo `LEFT OUTER JOIN`, onde podemos observar que a tabela da esquerda (DEPT) é a tabela onde pode não existir correspondência, ou seja, nossa tabela LEFT.

Em resumo o tipo de relacionamento `LEFT OUTER JOIN` nos diz: mesmo que não exista correspondência com a tabela da esquerda, traga seus dados, mesmo assim.

Exemplo de `RIGHT OUTER JOIN`:

```
SQL> select ename, dname
     from   dept right outer join emp
     on     emp.deptno = dept.deptno;

ENAME      DNAME
---------- --------------
FORD       RESEARCH
ADAMS      RESEARCH
SCOTT      RESEARCH
JONES      RESEARCH
SMITH      RESEARCH
JAMES      SALES
TURNER     SALES
BLAKE      SALES
MARTIN     SALES
WARD       SALES
ALLEN      SALES
MILLER
KING
CLARK

14 rows selected.

SQL>
```

Utilizando o mesmo exemplo, vamos trocar o tipo de relacionamento. Agora vamos relacionar as tabelas através de um `RIGHT OUTER JOIN`. Note que continuamos recuperando todos os empregados com seus respectivos de-

partamentos e localizações. Contudo, veja que os empregados que não estão alocados em departamento algum, também são retornados pelo comando `SELECT`. Voltando a analisar o comando, estamos selecionando os dados das tabelas DEPT e EMP, mas, agora, relacionando-as através de um relacionamento do tipo `RIGHT OUTER JOIN`, onde a tabela da direita (EMP) é a tabela onde pode não existir correspondência, ou seja, nossa tabela RIGHT.

Em resumo o tipo de relacionamento `RIGHT OUTER JOIN` nos diz: mesmo que não exista correspondência com a tabela da direita, traga seus dados, mesmo assim. Como já comentado, dentro do padrão ANSI ainda temos uma forma de relacionamento que é chamada de `FULL OUTER JOIN`. Para esta forma de relacionamento não há uma similar no Oracle. Ela consiste em buscar as linhas que possuem ligações entre as tabelas (`INNER JOIN`), mas também as que não existem entre as duas (`LEFT OUTER JOIN` e `RIGHT OUTER JOIN`), sem resultar em um Produto Cartesiano.

Contudo, em versões mais recentes, o Oracle Server aceita o padrão ANSI. Veja o exemplo a seguir.

```
SQL> select ename, dept.deptno, loc
     from emp full outer join dept
     on emp.deptno = dept.deptno
     /

ENAME          DEPTNO LOC
---------- ---------- -------------
SMITH              20 DALLAS
ALLEN              30 CHICAGO
WARD               30 CHICAGO
JONES              20 DALLAS
MARTIN             30 CHICAGO
BLAKE              30 CHICAGO
CLARK
SCOTT              20 DALLAS
KING
TURNER             30 CHICAGO
ADAMS              20 DALLAS
JAMES              30 CHICAGO
FORD               20 DALLAS
```

```
MILLER
                    40 BOSTON
                    10 NEW YORK

16 rows selected.

SQL>
```

Conforme pode ser visto, o `SELECT` retornou todos os empregados e departamentos, mesmo os que não são equivalentes.

Em resumo o tipo de relacionamento `FULL OUTER JOIN` nos diz: mesmo que não exista correspondência com a tabela da direita ou com a tabela da esquerda, traga seus respectivos dados, mesmo assim.

SELF JOIN

Este tipo de relacionamento indica que uma tabela também pode se relacionar com ela mesma. Em alguns casos podemos ter tabelas que possuem mais de um tipo de informação. Um exemplo é a tabela EMP. Nela temos informações de empregados, mas também de gerentes associados, onde, através da coluna MGR, nós conseguimos ver os gerentes de cada empregado. Note que os códigos dos gerentes inseridos nesta coluna são os que também aparecem na coluna EMPNO, ou seja, são empregados, também. Veja o exemplo:

```
SQL> select e.ename empregado, g.ename gerente
     from emp e, emp g where e.mgr = g.empno order by 1;

EMPREGADO  GERENTE
---------- ----------
ADAMS      SCOTT
ALLEN      BLAKE
BLAKE      KING
CLARK      KING
FORD       JONES
JAMES      BLAKE
JONES      KING
MARTIN     BLAKE
MILLER     CLARK
```

```
SCOTT       JONES
SMITH       FORD
TURNER      BLAKE
WARD        BLAKE

13 rows selected.

SQL>
```

Note que para usarmos a tabela EMP duas vezes na cláusula `FROM`, sem ocorrer o erro de ambiguidade entre as colunas, nós utilizamos um `ALIAS` diferente para cada uma delas.

Padrão ANSI SQL92 – SELF JOIN

Para unir uma tabela com ela mesma, criando assim, uma SELF JOIN,também existe uma forma dentro do padrão ANSI.Crie um relacionamento SELF JOIN através dacláusula ON.

```
SQL> select worker.last_name emp, manager.last_name mgr
   from employees worker JOIN employees manager
   on (worker.manager_id = manager.employee_id);

EMP                         MGR
-------------------------  -------------------------
Hartstein                   King
Zlotkey                     King
Cambrault                   King
Errazuriz                   King
Partners                    King
Russell                     King
Mourgos                     King
Vollman                     King
...
Hutton                      Zlotkey
Abel                        Zlotkey
Fay                         Hartstein
Gietz                       Higgins
```

```
106 rows selected.

SQL>
```

Produto Cartesiano ou Produto Cruzado

Sempre quando informamos mais de uma tabela na cláusula FROM, devemos, obrigatoriamente, colocar os relacionamentos referentes a estas tabelas. Caso isto não seja feito, o Oracle não gera uma mensagem de erro. Ao invés disso, ele gera um resultado o qual chamamos de "Produto Cartesiano" ou "Produto Cruzado". Imaginemos duas tabelas em um comando `SELECT`. A tabela "A" com 10 registros e a tabela "B" com 20 registros. Caso não seja informado um relacionamento na cláusula `WHERE`, e seja executado este comando SQL, o resultado será o retorno de 200 linhas, ou seja, para cada linha da tabela "A", ele retornará todas as linhas da tabela "B", ou vice-versa. Exemplo:

```
SQL> select ename, dname
     from emp, dept
     where emp.deptno = 20;

ENAME      DNAME
---------- --------------
SMITH      ACCOUNTING
SMITH      RESEARCH
SMITH      SALES
SMITH      OPERATIONS
JONES      ACCOUNTING
JONES      RESEARCH
JONES      SALES
JONES      OPERATIONS
SCOTT      ACCOUNTING
SCOTT      RESEARCH
SCOTT      SALES
SCOTT      OPERATIONS
ADAMS      ACCOUNTING
ADAMS      RESEARCH
ADAMS      SALES
```

```
ADAMS      OPERATIONS
FORD       ACCOUNTING
FORD       RESEARCH
FORD       SALES
FORD       OPERATIONS

20 rows selected.

SQL>
```

Para cada linha encontrada na tabela DEPT foi retornada todas as linhas da tabela de EMP.

Padrão ANSI SQL92 – Produto Cartesiano ou Produto Cruzado

Para gerar produto cartesiano ou produto cruzado, como também é chamado, utilizamos a cláusula CROSS JOIN.

```
SQL> select last_name, department_name
   from employees
   cross join departments;

LAST_NAME                 DEPARTMENT_NAME
------------------------- -----------------------
Abel                      Administration
Ande                      Administration
Atkinson                  Administration
Austin                    Administration
...
Vishney                   Payroll
Vollman                   Payroll
Walsh                     Payroll
Weiss                     Payroll
Whalen                    Payroll
Zlotkey                   Payroll

2889 rows selected.

SQL>
```

ALIAS de Tabela e Coluna

`ALIAS` é um apelido que podemos dar para uma tabela ou coluna no momento que estivermos escrevendo um comando SQL. Os `ALIAS` de tabela são comumente utilizados em SELECTS quando trabalhamos com mais de uma tabela na cláusula `FROM`. Quando isto acontece precisamos criar os relacionamentos na cláusula WHERE identificando cada coluna destes relacionamentos. Por convenção, na grande maioria das vezes, as colunas de relacionamento possuem o mesmo nome entre as tabelas para facilitar o entendimento do modelo de dados. Com isto, ao montar o comando `SELECT` precisamos dizer de quais colunas estamos nos referindo. Como os nomes de tabelas geralmente são nomes longos, para facilitar a identificação e a que se referem, ao invés de colocarmos o nome da tabela na frente das colunas, nós usamos um `ALIAS`. Usualmente o `ALIAS` é um nome pequeno formado por poucos caracteres, geralmente formado pelas iniciais do nome da própria tabela, gerando assim um rótulo para a mesma. Sua utilidade é simplesmente criar uma identificação para o objeto, auxiliando na construção de um comando visualmente mais enxuto.

```
SQL> select emp.first_name
           ,emp.email
           ,emp.job_id
           ,dep.department_name
     from   employees   emp
           ,departments dep
     where emp.department_id = dep.department_id
     and dep.department_id = 50
     and emp.manager_id    = 100;

FIRST_NAME           EMAIL            JOB_ID     DEPARTMENT_NAME
-------------------- ---------------- ---------- ---------------
Matthew              MWEISS           ST_MAN     Shipping
Adam                 AFRIPP           ST_MAN     Shipping
Payam                PKAUFLIN         ST_MAN     Shipping
Shanta               SVOLLMAN         ST_MAN     Shipping
Kevin                KMOURGOS         ST_MAN     Shipping

SQL>
```

Não diferente do uso para tabelas, o ALIAS de coluna também é utilizado para criar uma identificação. Contudo, na grande maioria das vezes não é usada para simplificar o nome e sim para clarificar seu significado. O ALIAS de coluna pode ser gerado de duas formas, utilizando ou não aspas duplas (“”). Quando queremos mostrar uma identificação mais formatada, podemos usar aspas duplas para isto. Por exemplo, temos a coluna DEPTNO que é o código do departamento. Se quisermos mostrar uma identificação formatada, ou seja, mais significativa, podemos declarar o ALIAS da coluna como “Código do Departamento”. Ao declararmos desta forma o Oracle mostra esta descrição como sendo o nome da coluna ao mostrar o resultado do SELECT. Se optarmos por não utilizar aspas para identificar a coluna, não podemos, por exemplo, ter descrições separadas por espaços em branco. Isto causaria confusão, dando a impressão de estarmos informando vários ALIAS para a mesma coluna. Neste caso, poderíamos fazer desta forma, CODIGO_DO_DEPARTAMENTO.

Apelidos de colunas

Para finalizar, você pode perceber que em alguns casos poderá ver a expressão `AS` antecedendo o apelido da coluna. Contudo, a utilização ou não desta expressão não fará nenhuma diferença. Neste caso, você pode optar por utilizá-la ou não, pois o resultado será o mesmo. Veja um exemplo:

```
SQL> select last_name AS "Último Nome"
           ,job_id AS cargo
           ,salary AS "Salário"
     from   employees
     where  (job_id = 'SA_REP'
     or     job_id = 'AD_PRES')
     and    salary > 15000
     /

Último Nome               CARGO         Salário
------------------------- ---------- ----------
King                      AD_PRES      35138,40
Vishney                   SA_REP       15373,05
Ozer                      SA_REP       16837,15
Abel                      SA_REP       16105,10
```

```
SQL>
```

Dessa forma o comando fica até um pouco mais legível.

Capítulo 8

Selecionando dados

8.1 Selecionando dados

SELECT

Com este comando, você seleciona linhas e colunas de uma ou mais tabelas.

Selecionando as colunas EMPNO e ENAME da tabela EMP:

```
SQL> select empno, ename from emp;

     EMPNO ENAME
---------- ----------
      7369 SMITH
      7499 ALLEN
      7521 WARD
      7566 JONES
```

```
      7654 MARTIN
      7698 BLAKE
      7782 CLARK
      7788 SCOTT
      7839 KING
      7844 TURNER
      7876 ADAMS
      7900 JAMES
      7902 FORD
      7934 MILLER

14 rows selected.

SQL>
```

SELECT *

O `*` substitui todas as colunas da tabela.

```
SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON

SQL>
```

ROWNUM

`ROWNUM` é uma pseudocoluna que indica um número sequencial referente às linhas retornadas que atendam a cláusula `WHERE` de um comando `SELECT`. Você também poderá utilizá-la na própria cláusula `WHERE` como forma de restrição. Neste caso, se no comando `SELECT` houver uma cláusula `ORDER BY`, a restrição será aplicada antes da ordenação. Veja alguns exemplos a seguir.

```
SQL> select rownum, deptno, dname from dept;
```

```
   ROWNUM     DEPTNO DNAME
--------- --------- --------------
        1         10 ACCOUNTING
        2         20 RESEARCH
        3         30 SALES
        4         40 OPERATIONS
        5         50 TI

SQL>
```

Nesse exemplo, a coluna `ROWNUM` traz um número sequencial para cada registro retornado.

```
SQL> select ename, job, deptno from emp order by deptno;

ENAME      JOB           DEPTNO
---------- --------- ---------
CLARK      MANAGER           10
KING       PRESIDENT         10
MILLER     CLERK             10
JONES      MANAGER           20
FORD       ANALYST           20
ADAMS      CLERK             20
SMITH      CLERK             20
SCOTT      ANALYST           20
WARD       SALESMAN          30
TURNER     VENDEDOR          30
ALLEN      SALESMAN          30
JAMES      CLERK             30
BLAKE      MANAGER           30
MARTIN     SALESMAN          30

14 rows selected.

SQL>

SQL> select ename, job, deptno from emp where
     rownum < 5 order by deptno;
```

```
ENAME      JOB        DEPTNO
---------- --------- ---------
SMITH      CLERK            20
JONES      MANAGER          20
WARD       SALESMAN         30
ALLEN      SALESMAN         30

SQL>
```

Note neste último exemplo que, primeiramente, foi aplicada a restrição para trazer os quatro primeiros registros, depois foi aplicada a ordenação. No `SELECT` anterior a este, vimos que, se fosse aplicada somente a ordenação dos dados, visualizando todo o resultado, poderíamos constatar que os quatro primeiros registros seriam todos do departamento 10 e 20. Agora veja todo o resultado sem ordenação.

```
SQL> select ename, job, deptno from emp;

ENAME      JOB        DEPTNO
---------- --------- ---------
SMITH      CLERK            20
ALLEN      SALESMAN         30
WARD       SALESMAN         30
JONES      MANAGER          20
MARTIN     SALESMAN         30
BLAKE      MANAGER          30
CLARK      MANAGER          10
SCOTT      ANALYST          20
KING       PRESIDENT        10
TURNER     VENDEDOR         30
ADAMS      CLERK            20
JAMES      CLERK            30
FORD       ANALYST          20
MILLER     CLERK            10

14 rows selected.

SQL>
```

ROWID

Cada linha armazenada no banco de dados tem um endereço. O banco de dados Oracle usa o tipo `ROWID` para armazenar o endereço (`ROWID`) de cada linha no banco de dados. O `ROWID` pode ser muito útil nos casos em que queremos identificar um valor único em determinada tabela, quando através das colunas nela existentes não conseguimos esta singularidade.

Este tipo se encaixa em três categorias:

- **ROWIDs físicos** (*Physical ROWIDs*): armazenam endereços de linhas de tabelas padrões do Oracle (*heap-organized tables*), tabelas clusterizadas (*clustered tables*) e tabelas e índices particionados. Estes IDs representam os endereços físicos, ou seja, levam em consideração as informações dos blocos de dados armazenados fisicamente no disco.
- **ROWIDs lógicos** (*Logical ROWIDs*): armazenam os endereços de linhas de tabelas indexadas (*index-organized tables*). Os índices b-TREE se baseiam nos `ROWIDs` lógicos para montar sua arvore e assim acelerar a busca pelos dados em uma consulta.
- **ROWIDs estrangeiros** (*Foreign ROWIDs*): são identificadores em tabelas estrangeiras, tais como tabelas do DB2 acessadas através de um gateway. Eles não são `ROWIDs` padrão do Oracle Database.

Segue um exemplo:

```
SQL> select rowid, d.* from dept d;

ROWID                  DEPTNO DNAME          LOC
------------------ --------- -------------- -------------
AAAAAAAAAAA                10 ACCOUNTING     NEW YORK
AAAAAAAAAAB                20 RESEARCH       DALLAS
AAAAAAAAAAC                30 SALES          CHICAGO
AAAAAAAAAAD                40 OPERATIONS     BOSTON
AAAAAAAAAAE                50 TI             BRASIL
AAAAAAAAAAF                60 ABC%           BRASIL

6 rows selected.
SQL>
```

Para cada linha retornada, existe um ROWID específico.

Tabela DUAL

DUAL é uma tabela que possui uma coluna chamada DUMMY, e um único registro com o valor X. Trata-se de uma tabela criada no SCHEMA SYS (que é seu dono) e que existe por padrão no banco Oracle. Ela é utilizada quando precisamos retornar um único valor, por exemplo, o valor de uma constante, expressão, retornar o resultado de cálculos numéricos e de datas ou pseudo-colunas ou seja, algo que não se origina de tabelas de dados comuns. Seu uso é muito comum em programas ou simples comandos SQL. Sua estrutura atende os requisitos da linguagem, pois contempla as cláusulas SELECT e FROM que são obrigatórias. Portanto, é uma tabela como outra qualquer, embora possua uma característica bem definida: a de auxiliar em determinadas operações, sendo desnecessário selecionar tabelas comuns de dados.

Selecionando os dados da tabela DUAL:

```
SQL> select * from dual;
D
-
X
SQL> select sysdate from dual;
SYSDATE
---------
13-APR-11

SQL>
```

Utilizando a tabela DUAL para cálculos com dados constantes, ou seja, não utilizando dados vindos de tabelas do banco de dados:

```
SQL> select 10*100/2 from dual;
 10*100/2
---------
      500
SQL> select mod(10,2) from dual;
MOD(10,2)
---------
```

```
         0
SQL>
```

8.2 SELECT FOR UPDATE

Ao executar um `SELECT` com a opção `FOR UPDATE`, estamos bloqueando as linhas da tabela selecionada. Como padrão, para o comando `SELECT`, o Oracle não faz este bloqueio, mas com o uso deste recurso ele muda a forma padrão de bloqueio. É muito útil quando queremos garantir que o que estamos visualizando em um dado momento não está sendo alterado.

Para liberar o bloqueio, precisamos executar um `COMMIT` ou `ROLLBACK`. Veja o exemplo a seguir.

```
SQL> select * from dept for update;
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     FLORIDA
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        43 ORDER MANAGER  BRASIL
        41 GENERAL LEDGER
        42 PURCHASING
6 linhas selecionadas.
SQL>
```

No exemplo, estamos bloqueando todas as linhas da tabela `DEPT`. Contudo, se neste comando SQL tivéssemos alguma cláusula `WHERE` restringindo o resultado, apenas estas linhas retornadas é que seriam bloqueadas. Vejo outro exemplo na sequência:

```
SQL> select employee_id, salary, commission_pct, job_id
     from   employees
     where  job_id = 'SA_REP'
     for update
     order  by employee_id
     /

EMPLOYEE_ID     SALARY COMMISSION_PCT JOB_ID
```

```
----------- ---------- -------------- ----------
        150      14641             .3 SA_REP
        151   13908.95            .25 SA_REP
        152    13176.9            .25 SA_REP
        155    10248.7            .15 SA_REP
        156      14641            .35 SA_REP
...
        176   12591.26             .2 SA_REP
        177   12298.44             .2 SA_REP
        178    10248.7            .15 SA_REP
        179    9077.42             .1 SA_REP

30 rows selected.

SQL>
```

Aqui estamos selecionando todos os empregados que têm como cargo `SA_REP`. Como estamos aplicando um `FOR UPDATE`, todas as linhas e, somente elas, estarão bloqueadas para qualquer alteração. Vale lembrar que, para liberá-las, precisamos executar um `COMMIT` ou `ROLLBACK`.

Você também pode bloquear linhas de mais de uma tabela. Isso é simples. Caso esteja aplicando um `FOR UPDATE` em um comando SQL em cuja cláusula há várias tabelas, automaticamente os registros listados serão bloqueados. A seguir, um exemplo:

```
SQL> select e.employee_id, e.salary, e.commission_pct
  2  from   employees e JOIN departments d
  3  USING  (department_id)
  4  where  job_id = 'ST_CLERK'
  5  and    location_id = 1500
  6  FOR UPDATE
  7  order  by e.employee_id
  8  /

EMPLOYEE_ID     SALARY COMMISSION_PCT
----------- ---------- --------------
        125    4685.12
        126    3953.07
        127    3513.84
```

```
        128    3221.02
...
        141    5124.35
        142    4538.71
        143    3806.66
        144    3660.25

20 rows selected.

SQL>
```

Note que aqui estamos selecionando dados de duas tabelas. Estamos selecionando empregados que possuem como cargo `ST_CLERK` e cuja localização seja `1500`. Logo, se tentássemos excluir dados, ou melhor, empregados cujo cargo fosse `ST_CLERK`, não conseguiríamos, pois estes registros estão bloqueados. O mesmo aconteceria se tentássemos excluir dados de departamento cuja localização fosse `1500`.

Caso você deseje bloquear apenas uma das tabelas, por exemplo, poderia aplicar o `FOR UPDATE` informando a coluna da tabela que você quer bloquear. Veja o mesmo exemplo anterior, agora informando a coluna da tabela a ser bloqueada.

```
SQL> select e.employee_id, e.salary, e.commission_pct
  2  from   employees e JOIN departments d
  3  using (department_id)
  4  where job_id = 'ST_CLERK'
  5  and   location_id = 1500
  6  FOR UPDATE of e.salary
  7  order by e.employee_id
  8  /

EMPLOYEE_ID     SALARY COMMISSION_PCT
----------- ---------- --------------
        125    4685.12
        126    3953.07
...
        140    3660.25
        141    5124.35
```

```
        142     4538.71
        143     3806.66
        144     3660.25

20 rows selected.

SQL>
```

DEADLOCKS

Com o uso do bloqueio de dados, sendo pela utilização do `FOR UPDATE`, `LOCK` ou qualquer outro meio, pode acontecer de transações ficarem bloqueadas de tal forma que uma acabe por ter que esperar pela outra. A isso se dá o nome de `DEADLOCK`. Por exemplo, uma transação "A" bloqueia determinados registros. Outra transação "B" tenta atualizá-los. Ela ficará esperando a transação "A" liberar os registros. Todavia, aconteceu que, antes de a transação "B" acessar tais registros, ela acabou atualizando outra fonte, que será utilizada pela transação "A" mais adiante. Quando a transação "A" for tentar acessar tal fonte, não conseguirá, pois "B" não efetivou as transações. Nisto, a transação "A" ficará esperando pela "B" e vice-versa. Geralmente, no banco de dados, há regras definidas para resolver este conflito, como por exemplo, tempo de espera por um recurso.

8.3 Selecionando o destino

FROM

Especifica o nome (ou nomes) da tabela de onde devem ser selecionadas as linhas.

Selecionando dados das tabelas `EMP` e `DEPT`:

```
SQL> select ename, dname
     from   emp e, dept d
     where  e.deptno = d.deptno
     and    d.deptno = 20;
ENAME      DNAME
---------- --------------
```

```
SMITH      RESEARCH
JONES      RESEARCH
SCOTT      RESEARCH
ADAMS      RESEARCH
FORD       RESEARCH
SQL>
```

Dentro da cláusula `FROM`, é possível escrever comandos `SELECT`, fazendo com que ele faça o papel de uma tabela ou visão.

```
SQL> select ename, dname
     from   emp e
           ,(select dname, deptno from dept) d
     where  e.deptno = d.deptno
     and    e.deptno = 20;

ENAME      DNAME
---------- --------------
SMITH      RESEARCH
JONES      RESEARCH
SCOTT      RESEARCH
ADAMS      RESEARCH
FORD       RESEARCH

SQL>
```

Note que estão sendo selecionados dados da tabela `EMP` e da tabela `DEPT`. Contudo, a tabela `DEPT` não está sendo acessada diretamente, mas, sim, através de um `SELECT`, como se fosse uma *view*. Nestes casos, é possível restringir colunas e dados, se necessário.

8.4 Restringindo dados

WHERE

Especifica a condição ou condições que as linhas devem satisfazer. A cláusula `WHERE` é utilizada como uma espécie de filtro para seus comandos SQL. É ela quem vai restringir a busca dos seus dados.

```
SQL> select first_name, email, country_name
     from   employees e
           ,departments d
           ,locations l
           ,countries c
     where e.department_id = d.department_id
     and   d.location_id   = l.location_id
     and   l.country_id    = c.country_id
     and   c.country_name  = 'Canada'
     /

FIRST_NAME           EMAIL                     COUNTRY_NAME
-------------------- ------------------------- ------------
Michael              MHARTSTE                  Canada
Pat                  PFAY                      Canada

SQL>
```

Neste exemplo, estamos selecionando dados de várias tabelas, utilizando a cláusula `WHERE` para fazer as ligações entre elas e também para restringir registros.

8.5 Escolhendo linhas e colunas

Através do comando `SELECT`, podemos selecionar apenas as linhas ou colunas que desejarmos.

```
SQL> select first_name, phone_number
     from   employees
     where  rownum < 5;

FIRST_NAME           PHONE_NUMBER
-------------------- --------------------
Steven               515.123.4567
Neena                515.123.4568
Lex                  515.123.4569
Alexander            515.423.4567

SQL>
```

8.6 Utilizando operadores

Algumas vezes, é preciso refinar a consulta para se obter o resultado esperado. Neste caso, o uso de filtros é imprescindível para que possam ser feitas restrições nos dados que serão buscados no banco de dados. Os filtros são definidos no SQL com a cláusula `WHERE`. Para representar estas condições, existem os operadores relacionais e os operadores lógicos.

Operadores relacionais:

- >: maior que
- <: menor que
- =: igual a
- <=: menor e igual a
- >=: maior e igual a
- **<>ou !=**: diferente de

Operadores lógicos:

- **AND** : operador `E`, utilizado para validar mais de uma condição;
- **OR** : operador `OU`, utilizado para validar pelo menos uma condição;
- **NOT** : representa negação de uma sentença;
- **[NOT] BETWEEN**: utilizado na restrição de intervalos de valores e datas;
- **[NOT] LIKE**: útil na pesquisa aproximada por sua característica de permitir caracteres curingas. O `%` vai substituir uma cadeia de caracteres, não importando quantos nem quais, e o `_` vai substituir um caractere apenas, não importa qual;
- **[NOT] IN** : definem condições com conjunto de valores;

- **IS [NOT] NULL** : verifica se o campo é nulo ou não. Sempre utilize este operador para verificar se o campo é ou não nulo. Nunca utilize o operador = para isto, como `campo = ""`; `campo = " "`; `campo = null` ou qualquer outra forma;

- **ANY** : compara o valor com cada valor resultante de uma subquery. Deve ser precedido pelos operandos `(,)`, >=, <=, !=;

- **[NOT] EXISTS** : retorna um resultado verdadeiro se uma subquery apresenta no mínimo uma linha dentro das condições da query.

Existe uma particularidade sobre o uso do `LIKE`. Às vezes, temos a necessidade de pesquisar por caracteres que fazem parte de alguma expressão ou sintaxe dentro da linguagem. Quando isso acontece, não conseguimos realizar tal pesquisa, pois na maioria das vezes é gerado um erro ou o resultado não é apresentado como queríamos. Para suprir essa necessidade, podemos definir `ESCAPES` para dizer à linguagem que naquele momento estes caracteres não representam funções, mas simples caracteres. Veja o exemplo:

```
SQL> insert into dept values(60,'ABC%','BRASIL');

1 row created.

SQL> select dname from dept where dname like '%%%';

DNAME
--------------
ACCOUNTING
RESEARCH
SALES
OPERATIONS
TI
ABC%

6 rows selected.

SQL> select * from dept where dname like '%/%%' escape '/';
```

```
DEPTNO    DNAME          LOC
--------- -------------- -------------
60        ABC%           BRASIL

SQL>
```

Neste exemplo, na cláusula onde aparece a função `LIKE`, podemos ver três símbolos de porcentagem. Sabemos que em SQL este símbolo é utilizado como expressão para determinar que o caractere especificado pode estar no início, fim ou em qualquer parte da `string`. O primeiro e o último símbolo de porcentagem referem-se às expressões utilizadas com o `LIKE`, já o segundo caractere de porcentagem é nosso parâmetro de pesquisa. Note que, para que a linguagem não confunda este caractere com uma expressão, ela é precedida pelo caractere de escape (`/`), que indica qual texto desejamos pesquisar. Este símbolo é definido através da cláusula `ESCAPE` e pode ser qualquer caractere exceto `%` e `_` (porcentagem e underscore).

Operador de concatenação

Além dos operadores exibidos na tabela, também temos mais um que se encaixa na categoria de operadores de concatenação. Ele é representado por dois pipelines, `||`.

Operador de Cotação (q) alternativo

O operador de cotação (q) é utilizado para substituir o delimitador de aspas utilizado nos comandos SQL, geralmente, para comparação de strings. Sabemos que para delimitar uma string utilizamos o caracter de aspas simples. Contudo, há casos onde a string utilizada contém uma ou mais aspas na sua composição. Por exemplo, Car's Paul, veja que esta string possui uma aspa simples na sua escrita. Desta forma, se fossemos utilizar esta string em uma comparação em um comando select ou se fosse utilizar em uma concatenação, faríamos desta forma: 'Car's Paul'. Veja o exemplo abaixo sendo executado.

O operador de cotação (q) é utilizado para substituir o delimitador de aspas utilizado nos comandos SQL, geralmente, para comparação de strings. Sabemos que para delimitar uma string utilizamos o caracter de aspas simples. Contudo, há casos onde a string utilizada contém uma ou mais aspas na sua composição. Por exemplo, Car's Paul, veja que esta string possui uma aspa simples na sua escrita. Desta forma, se fossemos utilizar esta string em uma

comparação em um comando select ou se fosse utilizar em uma concatenação, faríamos desta forma: 'Car's Paul'. Veja o exemplo abaixo sendo executado.

```
SQL> select 'Car's Paul' from dual
  2  /

select 'Car's Paul' from dual

ORA-01756: quoted string not properly terminated

SQL>
```

Note que ao executar o comando, um erro de sintaxe foi gerado, informando que a string de cotação não foi finalizada corretamente. Sabemos que a primeira e terceira aspas fazem parte da sintaxe para delimitar os caracteres. Contudo, a aspa do meio, que faz parte do contexto da string, ou melhor, da frase, está gerando o erro. Agora vamos substituir o caracter delimitador, através do comando de cotação q, que por padrão é aspa, por colchetes.

```
SQL> select q'[Car's Paul]' from dual
  2  /

Q'[CAR'SPAUL]'
--------------
Car's Paul
 s
SQL>
```

Note que agora, após definirmos outro delimitador, o comando SQL funcionou corretamente. Outros delimitadores podem ser utilizados conforme a conveniência. Os mesmos podem ser definidos com um ou vários bytes, ou qualquer um dos seguintes pares de caracteres: {}, () ou <>.

Regras de Precedência de Operadores

Dentro da linguagem SQL existe uma prioridade que define qual operador é executado por primeiro, ou seja, qual a precedência de cada operador dentro de uma instrução SQL.

1) Operadores aritiméticos

2) Operador de concatenação

3) Condições de comparação

4) IS [NOT] NULL, LIKE, [NOT] IN

5) [NOT] BETWEEN

6) Diferente de

7) Condição lógica NOT

8) Condição lógica AND

9) Condição lógica OR

Conforme pode ser visto neste quadro, existe uma precedência de execução para os operadores no momento que uma instrução SQL é executada. Portanto, quando um comando SQL é enviado para a execução, o motor do banco de dados ao avaliar questões, como, por exemplo, sintaxes e existência das tabelas e colunas envolvidas, ele também verifica esta tal precedência para determinar quais passos e em que ordem estes passos dever ser executados. Sabemos que um comando SQL tem como objetivo levar a um resultado, contudo, temos que ter em mente que o banco de dados não processa tudo de uma só vez, ele analisa cada parte da instução e vai gerando os resultados parciais até chegar a um resultado final. Desta forma, quando a instrução SQL for avaliada, o banco de dados tomará como ponto de partida esta precedência.

Você deve estar se perguntando qual a relevância disto na escrita de comandos SQL. Pois bem, se levarmos em conta questões de performance para comandos SQL complexos, ter o conhecimento de como agem estas regras de precedência, pode ajudá-lo a escrever um comando mais performático, digamos assim. Segundo as regras de otimização do banco de dados Oracle, uma instrução SQL pode ser modificada pelo próprio Otimizador do banco, visando uma melhor execução. Como? Escrevendo cláusulas comparativas levando em conta a ordem de precedência. Desta forma, o otimizador do banco de dados não precisará fazer este trabalho de reordenar estas cláusulas. Contudo, isto será revelevante quando estivermos falando de ajustes finos de

performance, ou seja, seu SQL não vai deixar de executar por falta da aplicação destas regras.

No entanto, você precisa conhecer estas regras, para que em determinados casos, não acabe comentendo enganos no momento que estiver escrevendo instruções SQL, como, por exemplo, quando estiver utilizando os operadores OR e AND. Veja o exemplo a seguir.

```
SQL> select last_name, job_id, salary
  2  from   employees
  3  where  job_id = 'SA_REP'
  4  or     job_id = 'AD_PRES'
  5  and    salary > 15000
  6
SQL>
```

Para entender este comando, vamos avaliá-lo utilizando as regras de precedência. Utilizando estas premisas podemos observar que neste comando temos duas condições. A primeira nos diz que devem ser selecionados empregados que possuem JOB_ID = 'AD_PRES' e que SALARY > 15000. A segunda é que o JOB_ID = 'SA_REP'. Note que estamos seguindo as regras de precedência, e a regra é clara, operadores AND tem precedência sobre operadores OR. Neste caso, serão selecionados os registros que satisfazerem a primeira condição, empregados com cargo igual 'AD_PRES' e salário maior que 15.000, ou a segunda condição, empregados com o cargo igal a 'SA_REP'. Para melhor entendermos, poderíamos visualiar as condições da seguinte forma:

```
job_id = 'SA_REP'
```

ou

```
job_id = 'AD_PRES' E salary > 15000
```

Agora, vamos ver o resultado:

```
LAST_NAME                 JOB_ID         SALARY
------------------------- ---------- ----------
King                      AD_PRES      35138,40
Tucker                    SA_REP       14641,00
```

```
Bernstein                 SA_REP       13908,95
Hall                      SA_REP       13176,90
Olsen                     SA_REP       11712,80
Cambrault                 SA_REP       10980,75
Tuvault                   SA_REP       10248,70
King                      SA_REP       14641,00
Sully                     SA_REP       13908,95
McEwen                    SA_REP       13176,90
Smith                     SA_REP       11712,80
Doran                     SA_REP       10980,75
Sewall                    SA_REP       10248,70
Vishney                   SA_REP       15373,05
...
Taylor                    SA_REP       12591,26
Livingston                SA_REP       12298,44
Grant                     SA_REP       10248,70
Johnson                   SA_REP        9077,42

31 rows selected

SQL>
```

Note que o resultado satisfez as condições da instrução SQL, trazendo apenas um registro para a primeira condição e outros trinta referentes a segunda condição.

Agora vamos analisar o comando SQL de outro ângulo. Vamos imaginar que na verdade nossa intenção não seria trazer o resultado da forma como foi mostrado, idéia seria trazer empregados que tivessem como cargo 'AD_PRES' ou 'SA_REP' e que o salário fosse maior que 15.000. Neste caso, da forma como foi escrito nosso comando, não vai satisfazer nossa condição, pois, como vimos, o comando SQL foi avaliado e executado conforme as regras de precendência de operadores. Com isto, para fazer com que nossa condição seja satisfeita, vamos sobrepor a precedência padrão da linguagem.

Mesmo existindo estas regras de precedência para os operadores, é possível, sim, sobrepo-las com a ajuda de parênteses. Vamos, novamente, observar o comando SQL, agora alterado para satisfazer nossa condição:

```
SQL> select last_name, job_id, salary
```

```
  2  from    employees
  3  where   (job_id = 'SA_REP'
  4  or      job_id = 'AD_PRES')
  5  and     salary > 15000
  6  /
SQL>
```

Analisando o comando SQL, note que apenas acrescentamos um par de parênteses isolando duas de nossas comparações. Com isto, formamos duas novas condições. A primeira condição irá recuperar os empregados onde JOB_ID = 'SA_REP' ou JOB_ID = 'AD_PRES'. Já a segunda condição irá trazer os empregados onde SALARY > 15000. Desta forma, estamos sobrepondo a precedência dos operadores, forçando com que a primeira condição, mesmo contendo um operador OR, seja resolvida primeiro, para então, a segunda condição possa ser resolvida. Desta forma, poderíamos visualizá-las da seguinte forma:

```
(job_id = 'SA_REP' OU job_id = 'AD_PRES')
```

E

```
salary > 15000
```

Vamos ver o resultado abaixo:

```
LAST_NAME                 JOB_ID         SALARY
------------------------- ---------- ----------
King                      AD_PRES      35138,40
Vishney                   SA_REP       15373,05
Ozer                      SA_REP       16837,15
Abel                      SA_REP       16105,10

SQL>
```

Como pode ser visto, o resultado atendeu as expectativas. Foram listados todos os empregados que possuem o cargo igual a 'AD_PRES' ou 'SA_REP', e que possuem salário maior que 15.000.

Anteriormente, vimos os operadores que podem ser utilizados dentro da linguagem SQL. Vejamos alguns exemplos.

Utilizando o operador relacional `MAIOR QUE`:

```
SQL> select ename, sal
     from   emp
     where  sal> 2000;

ENAME              SAL
---------- ---------
JONES             2975
BLAKE             2850
CLARK             2450
SCOTT             3000
KING              5000
FORD              3000

6 rows selected.

SQL>
```

Utilizando os operadores relacionais `MAIOR QUE` e de `IGUALDADE`:

```
SQL> select ename, sal
     from   emp
           ,dept
     where  sal> 2000
     and    emp.deptno = dept.deptno
     and    dept.loc = 'NEW YORK';

ENAME              SAL
---------- ---------
CLARK             2450
KING              5000

SQL>
```

Utilizando os operadores relacionais `DIFERENÇA` e de `IGUALDADE`:

```
SQL> select count(*)
     from emp
     where deptno = 30;

COUNT(*)
```

```
---------
        6

SQL>
SQL> select count(distinct(empno))
     from emp
     where deptno<> 20;

COUNT(DISTINCT(EMPNO))
----------------------
                     9

SQL>
```

Utilizando `BETWEEN` para restringir datas:

```
SQL> select count(*)
     from emp
     where hiredate between '01-JAN-1980' and '01-DEC-1982';

COUNT(*)
---------
       12

SQL>
```

O uso do operador `IN` para restringir por mais de um conjunto de strings:

```
SQL> select dname
     from dept
     where loc in ('DALLAS','CHICAGO');

DNAME
--------------
RESEARCH
SALES

SQL>
```

Nestes exemplos, está sendo utilizado o operador `LIKE` para a realização de restrições, filtrando por partes de caracteres em uma coluna:

```
SQL> select job_id, job_title
     from jobs
     where lower(job_id) like 'mk%';

JOB_ID     JOB_TITLE
---------- -----------------------------------
MK_MAN     Marketing Manager
MK_REP     Marketing Representative

SQL>

SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
       50 TI             BRASIL

SQL> select dname from dept where dname like '%_S_%';

DNAME
--------------
RESEARCH

SQL>

SQL> select ename, job, hiredate
     from emp
     where to_char(hiredate,'dd/mm/yyyy') like '03/__/%';

ENAME      JOB       HIREDATE
---------- --------- ---------
JAMES      CLERK     03-DEC-81
FORD       ANALYST   03-DEC-81

SQL>
```

Agora o uso do `EXISTS` e `NOT EXISTS`, como forma de apenas verificar a existência ou não de determinados registros em outras tabelas. Isto é usado para quando não necessitamos retornar dados destas tabelas, apenas verificar correspondências entre elas:

```
SQL> select dname
     from dept d
     where  exists (select 1
                    from emp e
                    where e.deptno = d.deptno
                    and e.hiredate between '01-JUL-1980'
                      and '31-DEC-1980');

DNAME
--------------
RESEARCH

SQL>

SQL> select dname
     from dept d
     where not exists (select 1
                       from emp e
                       where e.deptno = d.deptno);

DNAME
--------------
OPERATIONS

SQL>
```

Por último, neste exemplo, temos o uso dos dois *pipelines* para a concatenação de dados:

```
SQL> select ename, 'R$ '||sal from emp where deptno = 10;

ENAME      'R$'||SAL
---------- -------------------------------------------
CLARK      R$ 2450
```

```
KING         R$ 5000
MILLER       R$ 1300

SQL>
```

Capítulo 9

Manipulação de dados

9.1 Inserção de dados

INSERT

Este comando inclui linhas em uma tabela ou visão. Vejamos o exemplo:

```
SQL> select * from emp;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- ---------- ----------
      7369 SMITH      CLERK           7902 17-DEC-80        800
      7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
      7521 WARD       SALESMAN        7698 22-FEB-81       1250
      7566 JONES      MANAGER         7839 02-APR-81       2975
      7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
```

```
     7698 BLAKE      MANAGER        7839 01-MAY-81       2850
     7782 CLARK      MANAGER        7839 09-JUN-81       2450
     7788 SCOTT      ANALYST        7566 09-DEC-82       3000
     7839 KING       PRESIDENT           17-NOV-81       5000
     7844 TURNER     VENDEDOR       7698 08-SEP-81       1500
     7876 ADAMS      CLERK          7788 12-JAN-83       1100
     7900 JAMES      CLERK          7698 03-DEC-81        950
     7902 FORD       ANALYST        7566 03-DEC-81       3000
     7934 MILLER     CLERK          7782 23-JAN-82       1300

      COMM     DEPTNO
 --------- ---------
                  20
       300        30
       500        30
                  20
      1400        30
                  30
                  10
                  20
                  10
         0        30
                  20
                  30
                  20
                  10


14 rows selected.

SQL>
```

Vamos inserir um registro na tabela EMP:

```
SQL> insert into emp(empno, ename, job, mgr, hiredate, sal,
     comm, deptno) values(8000,'JOHN','CLERK',7902,'30-MAR-2011'
     ,1000,200,20);

1 row created.
```

```
SQL> commit;

Commit complete.

SQL>
```

Veja o resultado a seguir:

```
SQL> select * from emp;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- --------- ---------- --------- ----------
      7369 SMITH      CLERK           7902 17-DEC-80        800
      7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
      7521 WARD       SALESMAN        7698 22-FEB-81       1250
      7566 JONES      MANAGER         7839 02-APR-81       2975
      7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
      7698 BLAKE      MANAGER         7839 01-MAY-81       2850
      7782 CLARK      MANAGER         7839 09-JUN-81       2450
      7788 SCOTT      ANALYST         7566 09-DEC-82       3000
      7839 KING       PRESIDENT            17-NOV-81       5000
      7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
      7876 ADAMS      CLERK           7788 12-JAN-83       1100
      7900 JAMES      CLERK           7698 03-DEC-81        950
      7902 FORD       ANALYST         7566 03-DEC-81       3000
      7934 MILLER     CLERK           7782 23-JAN-82       1300
      8000 JOHN       CLERK           7902 30-MAR-11       1000

      COMM     DEPTNO
 --------- ---------
                  20
       300        30
       500        30
                  20
      1400        30
                  30
                  10
                  20
                  10
         0        30
```

```
                    20
                    30
                    20
                    10
       200          20

15 rows selected.

SQL>
```

Agora, inserindo um novo departamento:

```
SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON

SQL> insert into dept(deptno
                       ,dname
                       ,loc)
               values ( 50
                       ,'TI'
                       ,'BRASIL')
     /

1 row created.

SQL> commit;

Commit complete.

SQL>
```

Veja o resultado a seguir:

```
SQL> select deptno, dname, loc from dept;
```

```
   DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 TI             BRASIL

SQL>
```

Inserindo dados através de SELECTS

Vimos anteriormente que é possível criar uma tabela com base em um `SELECT` em outra tabela. Este recurso também pode ser utilizado para o comando `INSERT`. Observe:

```
SQL> select * from clerk;

no rows selected

SQL> insert into clerk select * from emp where job = 'CLERK';

4 rows created.

SQL> select * from clerk;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- --------- ----------
      7369 SMITH      CLERK           7902 17-DEC-80        800
      7876 ADAMS      CLERK           7788 12-JAN-83       1100
      7900 JAMES      CLERK           7698 03-DEC-81        950
      7934 MILLER     CLERK           7782 23-JAN-82       1300

      COMM     DEPTNO
 ---------- ----------
                   20
                   20
                   30
                   10
```

```
SQL>
```

Neste exemplo, estamos alimentando a tabela `CLERK` com base nos dados vindos da tabela `EMP`.

9.2 Atualização de dados

Atuallizando dados através de UPDATE

O comando `UPDATE` muda o valor de um ou mais campos de uma tabela. Você pode alterar uma ou mais colunas por vez e também delimitar quais registros quer que o comando afete.

Quando ele é executado, para que a integridade dos dados seja mantida, o Oracle deixa locadas as linhas que foram afetadas pelo comando de atualização. Nesse caso, outros usuários não conseguirão atualizá-las, a menos, é claro, que você execute um `COMMIT` ou um `ROLLBACK`, finalizando a transação.

Para delimitar os registros, use sempre o comando juntamente com a cláusula `WHERE`. Caso esta cláusula não esteja informada no comando, todos os registros da tabela serão afetados por ele.

```
SQL> select ename, job, hiredate, sal, comm, deptno
     from emp
     where deptno = 30;

ENAME      JOB       HIREDATE        SAL       COMM     DEPTNO
---------- --------- --------- --------- ---------- ---------
ALLEN      SALESMAN  20-FEB-81      1600        300         30
WARD       SALESMAN  22-FEB-81      1250        500         30
MARTIN     SALESMAN  28-SEP-81      1250       1400         30
BLAKE      MANAGER   01-MAY-81      2850                    30
TURNER     VENDEDOR  08-SEP-81      1500          0         30
JAMES      CLERK     03-DEC-81       950                    30

6 rows selected.

SQL>
```

```
SQL> update emp
     set comm = comm + (sal/100*10)
     where hiredate between to_date('01/02/1981','dd/mm/yyyy')
                        and to_date('31/05/1981','dd/mm/yyyy');

4 rows updated.

SQL> select ename, job, hiredate, sal, comm, deptno
     from emp
     where deptno = 30;

ENAME      JOB       HIREDATE         SAL       COMM     DEPTNO
---------- --------- --------- ---------- ---------- ----------
ALLEN      SALESMAN  20-FEB-81       1600        460         30
WARD       SALESMAN  22-FEB-81       1250        625         30
MARTIN     SALESMAN  28-SEP-81       1250       1400         30
BLAKE      MANAGER   01-MAY-81       2850                    30
TURNER     VENDEDOR  08-SEP-81       1500          0         30
JAMES      CLERK     03-DEC-81        950                    30

6 rows selected.

SQL>
```

Aqui, estão sendo atualizadas as comissões dos empregados, com base em uma determinada faixa de datas de admissão.

Note que o funcionário `BLAKE` não sofreu alteração no salário, embora esteja dentro dos critérios de seleção. Isso se deve ao fato de que o Oracle não considera o cálculo envolvendo colunas com valores nulos. Para ele, valor nulo indica `FALSO`. Desta forma, qualquer cálculo envolvendo valores nulos, é ignorado para a linguagem. Vamos ver mais à frente que podemos contornar essa situação utilizando funções específicas.

Já no próximo exemplo, atualizações estão sendo feitas na tabela `LOCATIONS`. Note que estaremos utilizando uma subconsulta na cláusula `WHERE` para restringir os dados a serem atualizados.

```
SQL> select street_address, city, state_province, country_name
     from   locations l
```

```
           ,countries c
     where l.country_id = c.country_id
     and country_name in ('Italy','Canada', 'China');

STREET_ADDRESS                   CITY            STATE_PROVINCE
-------------------------------- --------------- ---------------
147 Spadina Ave                   Toronto         Ontario
6092 Boxwood St                   Whitehorse      Yukon
40-5-12 Laogianggen               Beijing
1297 Via Cola di Rie              Roma
93091 Calledel la Testa           VeniceItaly

COUNTRY_NAME
---------------
 Canada
 Canada
 China
 Italy

SQL>

SQL> update locations
     set state_province = 'S/P' -- sem província
     where country_id in (select l.country_id
                          from   locations l
                                ,countries c
                          where  l.country_id = c.country_id
                          and    country_name in ('Italy',
                                                  'Canada',
                                                  'China'))
      and state_province is null;

3 rows updated.

SQL> select street_address, city, state_province, country_name
     from   locations l
           ,countries c
     where l.country_id = c.country_id
     and country_name in ('Italy','Canada', 'China');
```

```
STREET_ADDRESS                    CITY            STATE_PROVINCE
-------------------------------- --------------- ---------------
147 Spadina Ave                  Toronto         Ontario
602 Boxwood St                   Whitehorse      Yukon
412 Laogianggen                  Beijing         S/P
127 Via Cola di Rie              Roma            S/P
9091 Calledel la Testa           Venice          S/P

 COUNTRY_NAME
---------------
Canada
Canada
China
Italy
Italy

SQL>
```

Atualizando dados através de SELECTS

Você também pode atualizar os dados através de SELECTS.

```
SQL> select * from clerk;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80       800
     7876 ADAMS      CLERK           7788 12-JAN-83      1100
     7900 JAMES      CLERK           7698 03-DEC-81       950
     7934 MILLER     CLERK           7782 23-JAN-82      1300

     COMM    DEPTNO
--------- ---------
                 20
                 20
                 30
                 10

SQL> update clerk
     set comm = (select comm from emp where empno = 7499);
```

```
4 rows updated.

SQL> select * from clerk;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
      300        20
      300        20
      300        30
      300        10

SQL>
```

Repare que a coluna de comissões, a tabela `CLERK`, está sendo atualizada com base no valor retornado da comissão, referente ao empregado cujo código é `7499`.

9.3 Exclusão de dados

Excluindo dados através de DELETE

Este comando é responsável por excluir linhas de uma tabela. Assim como o comando `UPDATE`, ele mantém locadas as linhas enquanto a transação não é concluída. Ele também pode ser usado juntamente com a cláusula `WHERE` para restringir registros. O cuidado com a exclusão de registros também vale para este comando, pois, sem isso, você pode excluir todos os dados de uma tabela.

```
SQL> select * from emp;
```

```
    EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- --------- ---------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81       2975
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300
     8000 JOHN       CLERK           7902 30-MAR-11       1000

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10
      200        20

15 rows selected.

SQL>
```

```
SQL> delete from emp where deptno = 20;

6 rows deleted.

SQL>
```

Excluímos todos os empregados cujo departamento era o `20`. Veja o resultado a seguir:

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7499 ALLEN      SALESMAN       7698 20-FEB-81      1600
     7521 WARD       SALESMAN       7698 22-FEB-81      1250
     7654 MARTIN     SALESMAN       7698 28-SEP-81      1250
     7698 BLAKE      MANAGER        7839 01-MAY-81      2850
     7782 CLARK      MANAGER        7839 09-JUN-81      2450
     7839 KING       PRESIDENT           17-NOV-81      5000
     7844 TURNER     VENDEDOR       7698 08-SEP-81      1500
     7900 JAMES      CLERK          7698 03-DEC-81       950
     7934 MILLER     CLERK          7782 23-JAN-82      1300

     COMM    DEPTNO
--------- ---------
      300        30
      500        30
     1400        30
                 30
                 10
                 10
        0        30
                 30
                 10


9 rows selected.

SQL>
```

Agora vamos excluir dados da tabela de departamento, dos quais não existe nenhuma referência na tabela de empregados. Note o uso do operador `NOT EXISTS` como auxílio na exclusão dos dados.

```
SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
       50 TI             BRASIL

SQL>

SQL> delete from dept d
     where not exists (select 1
                       from emp e
                       where e.deptno = d.deptno);

2 rows deleted.

SQL>
```

Veja o resultado em seguida:

```
SQL> select * from dept;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO

SQL>
```

Excluindo dados através de SELECTS

Podemos também utilizar cláusulas `SELECTS` para auxiliar na exclusão de dados:

```
SQL> delete from clerk
     where deptno = (select deptno from clerk where sal> 1200);

1 row deleted.

SQL>
```

Capítulo 10

Trabalhando com funções

10.1 Funções de caracteres, de cálculos e operadores aritméticos

Funções de caracteres e de cálculos também podem ser usadas nas expressões SQL. Através delas, podem-se modificar os dados, tanto no que diz respeito aos valores selecionados, como também na forma como são apresentados. Por exemplo, separar informações dentro de uma determinada `STRING`, concatenar caracteres, definir a caixa das letras (maiúsculas, minúsculas e intercaladas), efetuar arredondamentos em valores, calcular a média, calcular raiz quadrada etc.

Já os operadores aritméticos podem ser utilizados para a inserção de cálculos dentro dos comandos SQL. Cálculos, estes, referentes à soma, subtração, divisão e multiplicação.

Vale salientar que estas funções e operadores podem ser utilizados em qualquer cláusula SQL exceto na cláusula `FROM`.

Funções de caracteres

- **INITCAP:** retorna o primeiro caractere de cada palavra em maiúscula;
- **LOWER:** força caracteres maiúsculos aparecerem em minúsculo;
- **UPPER:** força caracteres minúsculos aparecerem em maiúsculo;
- **SUBSTR:** extrai um trecho de uma string, começando por uma posição inicial e a partir desta posição conta com base na quantidade solicitada;
- **TO_CHAR:** converte um valor numérico para uma string de caracteres. Também é utilizada para inserir máscara em campos numéricos e de data;
- **INSTR:** retorna a posição do primeiro caractere encontrado, passado como parâmetro;
- **LENGTH:** traz o tamanho dos caracteres em bytes;
- **RPAD:** faz alinhamento à esquerda e preenche com caracteres à direita, até uma determinada posição. Ambos os valores são passados como parâmetro;
- **LPAD:** faz alinhamento à direita e preenche com caracteres à esquerda, até uma determinada posição. Ambos os valores são passados como parâmetro.
- **CONCAT:** Utilizado para concatenar, ou seja, unir valores. Contudo é limitado a dois parâmetros.

Seguem exemplos do uso destas funções.

Primeiro, veja o uso das funções `INITCAP`, `LOWER` e `UPPER`. Note que o valores das colunas são alterados conforme a ação de cada função.

```
SQL> select ename from emp where job = 'MANAGER';

ENAME
----------
JONES
BLAKE
CLARK

SQL>

SQL> select initcap(ename) from emp where job = 'MANAGER';

INITCAP(EN
----------
Jones
Blake
Clark

SQL>

SQL> select dname, lower(dname) from dept;

DNAME          LOWER(DNAME)
-------------- --------------
ACCOUNTING     accounting
RESEARCH       research
SALES          sales
OPERATIONS     operations

SQL>

SQL> select upper('treinamento de sql') from dual;

UPPER('TREINAMENTO
------------------
TREINAMENTO DE SQL

SQL>
```

O próximo exemplo mostra o uso das funções SUBSTR e UPPER. O exemplo retorna os nomes das regiões mostrando apenas parte da string referente

a estes nomes:

```
SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa

SQL> select region_name, upper(substr(region_name,1,2))
     from   regions;

REGION_NAME                UPPER(SU
-------------------------- --------
Europe                     EU
Americas                   AM
Asia                       AS
Middle East and Africa     MI

SQL>
```

O próximo exemplo mostra o uso da função `TO_CHAR`. Neste exemplo, a função foi utilizada para formatar os valores da coluna `SAL` (salários):

```
SQL> select ename, sal,
            'R$ '||to_char(sal,'fm999G990D00') salario
     from emp
     where comm is null;

ENAME             SAL SALARIO
---------- ---------- --------------
SMITH             800 R$ 800,00
JONES            2975 R$ 2.975,00
BLAKE            2850 R$ 2.850,00
CLARK            2450 R$ 2.450,00
SCOTT            3000 R$ 3.000,00
KING             5000 R$ 5.000,00
ADAMS            1100 R$ 1.100,00
```

```
JAMES                950 R$ 950,00
FORD                3000 R$ 3.000,00
MILLER              1300 R$ 1.300,00

10 linhas selecionadas.

SQL>
```

Este exemplo mostra o uso da função `INSTR`:

```
SQL> select instr(37462.12,'62') from dual;

INSTR(37462.12,'62')
--------------------
                   4

SQL>
```

Este exemplo mostra o uso da função `LENGTH`:

```
SQL> select first_name
     from   employees
     where  length(first_name) > 10;

FIRST_NAME
--------------------
Christopher
Jose Manuel

SQL>
```

Este exemplo mostra o uso das funções `RPAD` e `LPAD`:

```
SQL> select rpad(last_name,12,'++++')
           ,lpad(salary,7,'0')
     from   employees
     where department_id = 30;

RPAD(LAST_NA LPAD(SA
------------ -------
```

```
Raphaely++++ 0011000
Khoo++++++++ 0003100
Baida+++++++ 0002900
Tobias++++++ 0002800
Himuro++++++ 0002600
Colmenares++ 0002500

6rowsselected.

SQL>
```

Por fim, o uso da função CONCAT:

```
SQL> select concat('Livro ','SQL') concatenado from dual
  2  /

CONCATENADO
-----------
Livro SQL

SQL>
```

Funções de cálculos

- **ROUND:** arredonda valores com casas decimais;
- **TRUNC:** trunca valores com casas decimais;
- **MOD:** mostra o resto da divisão de dois valores;
- **SQRT:** retorna a raiz quadrada de um valor;
- **POWER:** retorna um valor elevado a outro valor;
- **ABS:** retorna o valor absoluto;
- **CEIL:** retorna o menor inteiro, maior ou igual a valor;
- **FLOOR:** retorna o maior inteiro, menor ou igual a valor;

- **SIGN:** se valor maior que 0, retorna +1. Se valor menor que 0, retorna -1. Se valor igual a 0, retorna 0.

Os exemplos a seguir mostram o uso da função `ROUND`. Observe as diferentes formas de chamada a esta função.

```
SQL> select (sal / 2.7), round((sal / 2.7)), ename
     from emp
     where comm is not null;

 (SAL/2.7) ROUND((SAL/2.7)) ENAME
---------- ---------------- ----------
592,592593              593 ALLEN
462,962963              463 WARD
462,962963              463 MARTIN
555,555556              556 TURNER

SQL> select (sal / 2.7), round((sal / 2.7),2), sal,
     ename from emp where empno between 7500 and 7700;

 (SAL/2.7) ROUND((SAL/2.7),2)        SAL ENAME
---------- ------------------ ---------- ----------
462,962963             462,96       1250 WARD
1101,85185            1101,85       2975 JONES
462,962963             462,96       1250 MARTIN
1055,55556            1055,56       2850 BLAKE

SQL>
```

A próxima função exemplificada é a `TRUNC`. No exemplo, são mostradas diferentes formas de chamada a esta função.

```
SQL> select (salary / 2.7), trunc((salary / 2.7))
     from   employees
     where  email = 'NSARCHAN'
     /

(SALARY/2.7) TRUNC((SALARY/2.7))
------------ -------------------
```

```
  1555,55556                1555

SQL>
SQL> select (salary / 2.7), trunc((salary / 2.7),2)
     from   employees
     where employee_id between 100 and 105;

(SALARY/2.7) TRUNC((SALARY/2.7),2)
------------ ---------------------
  8888,88889               8888,88
   6296,2963               6296,29
   6296,2963               6296,29
  3333,33333               3333,33
  2222,22222               2222,22
  1777,77778               1777,77

6 linhas selecionadas.

SQL>
```

Estes exemplos mostram o uso das funções `MOD`, `SQRT` e `POWER`:

```
SQL> select mod(10,2) from dual;

MOD(10,2)
----------
0
SQL>

SQL> select sqrt(64) from dual;

SQRT(64)
----------
8
SQL>

SQL> select power(8,2) from dual;

POWER(8,2)
----------
```

```
         64
SQL>
```

A seguir é mostrado o uso das funções `ABS`, `CEIL` e `FLOOR`.

```
SQL> select abs(-20) from dual;

ABS(-20)
----------
        20

SQL>

SQL> select ceil(10.2) from dual;

CEIL(10.2)
----------
        11

SQL>

SQL> select floor(10.2) from dual;

FLOOR(10.2)
-----------
         10

SQL>
```

Por último, vejamos o uso da função `SIGN`. Nos exemplos, são mostrados diferentes parâmetros na chamada desta função.

```
SQL> select sign(-200) from dual;

SIGN(-200)
-----------
         -1

SQL> select sign(200) from dual;
```

```
SIGN(200)
----------
         1

SQL> select sign(0) from dual;

SIGN(0)
----------
0

SQL>
```

Operadores aritméticos

A seguir, veja os operadores em sua sequência de prioridade.

- * : multiplicação
- / : divisão
- + : adição
- - : subtração

Este exemplo mostra o uso das funções aritméticas de multiplicação e divisão:

```
SQL> select sal, (sal*2/3) from emp
     where comm is not null;

      SAL (SAL*2/3)
--------- ---------
     1600 1066.6667
     1250 833.33333
     1250 833.33333
     1500      1000
     1000 666.66667

SQL>
```

Este exemplo mostra o uso das funções aritméticas de multiplicação, divisão e soma:

```
SQL> select ename, sal, round((sal*2)/3+100.00,2)
     from emp
     where deptno = 30;

ENAME            SAL ROUND((SAL*2)/3+100.00,2)
---------- --------- -------------------------
ALLEN           1600                   1166.67
WARD            1250                    933.33
MARTIN          1250                    933.33
BLAKE           2850                      2000
TURNER          1500                      1100
JAMES            950                    733.33

6 rows selected.

SQL>
```

Este exemplo mostra o uso das funções aritméticas de multiplicação, divisão e subtração:

```
SQL> select ename
           ,dname
           ,trunc((sysdate - hiredate) / 365) anos
           ,hiredate
           ,(sal/10*trunc((sysdate - hiredate) / 365)) premiacao
           ,sal
     from   emp  e
           ,dept d
     where e.deptno = d.deptno
     and trunc((sysdate - hiredate) / 365) = 30;

ENAME      DNAME                     ANOS HIREDATEPREMIACAO
---------- -------------- ---------- -------- ----------
MILLER     ACCOUNTING                  30 23/01/82       3900
KING       ACCOUNTING                  30 17/11/81      15000
FORD       RESEARCH                    30 03/12/81       9000
```

```
JAMES       SALES                   30 03/12/81       2850
TURNER      SALES                   30 08/09/81       4500
MARTIN      SALES                   30 28/09/81       3750

     SAL
----------
      1300
      5000
      3000
       950
      1500
      1250

6 linhas selecionadas.

SQL>
```

10.2 Funções de agregação (grupo)

As funções de agregação são responsáveis por agrupar vários valores e retornar somente um único valor para um determinado grupo. As funções de agregação, também chamadas de funções de grupo, são especificadas no comando `SELECT` e são seguidas pela coluna à qual se aplicam. A utilização das funções de agregação pode implicar no uso da cláusula `GROUP BY`. Isso acontece porque, ao informarmos colunas com funções e colunas sem funções em um `SELECT`, precisamos agrupar as colunas que não estão sendo afetadas pelo agrupamento causado pelas funções. Veja a ilustração:

Empregado	Departamento	Salário
SMITH	RESEARCH	800
ALLEN	SALES	1600
WARD	SALES	1250
JONES	RESEARCH	2975
MARTIN	SALES	1250
BLAKE	SALES	2850
CLARK	ACCOUNTING	2450

Fig. 10.1

Repare na imagem que temos alguns empregados, departamentos e valores de salário. Nosso objetivo aqui é tentar de alguma forma somar todos os salários por departamento, ou seja, ver quanto de salário temos para os empregados referentes aos departamentos `RESEARCH`, `SALES` E `ACCOUNTING`. Da forma como os dados estão dispostos, não conseguimos visualizar isso, pois, se tentarmos agrupar por departamento, não conseguiremos, já que o agrupamento consiste em selecionar dados que possuem o mesmo valor e torná-lo único para cada conjunto de dados.

Por exemplo, temos os departamentos `RESEARCH`, `SALES` e `ACCOUNTING` aparecendo diversas vezes. Se agruparmos, teremos um único registro para o departamento `RESEARCH`, outro para `SALES`, e outro para `ACCOUNTING`. Entretanto, também estamos selecionando os nomes dos empregados e, na maioria dos casos, cada um possui um nome deferente, impossibilitando que os agrupemos. Se não conseguimos agrupar os empregados, não conseguimos agrupar os departamentos. É como se fosse uma sequência.

Quando usamos funções de grupo nas colunas de um `SELECT`, temos que agrupar todas as outras colunas, sendo através de uma função de agregação ou sendo pelo uso do `GROUP BY`. Já vimos que se nós quisermos a somatória de todos os salários por departamento não podemos estar selecionando os

empregados, ou melhor, seus nomes. Logo, a coluna `Empregado` não poderá aparecer no nosso `SELECT`. Caso contrário, estaríamos agrupando também por empregados, o que nos daria um agrupamento inútil, tendo em vista que cada nome de empregado é diferente.

Sempre quando trabalhamos com agrupamentos temos que ter em mente a seguinte situação: vai haver colunas que estarão sob o efeito das funções de agregação, como funções de somatória ou de média, e colunas que não estarão sobre o efeito destas funções, mas que precisarão ser agrupadas para que, juntas, possam formar um conjunto de dados.

Veja a próxima ilustração:

Agrupamento por Group By

Agrupamento por somatória

Empregado	Departamento	Salário
SMITH	RESEARCH	800
ALLEN	SALES	1600
WARD	SALES	1250
JONES	RESEARCH	2975
MARTIN	SALES	1250
BLAKE	SALES	2850
CLARK	ACCOUNTING	2450

Resultado final: Valores não agrupados por departamento

Fig. 10.2

Nesta outra ilustração temos dois grupos. Um, formado pelas colunas `Empregado` e `Departamento`, que sofrerão a ação do `GROUP BY`, e outro, formado apenas pela coluna `Salário`, que sofrerá a ação da nossa função de agregação. Vale ressaltar que nosso objetivo aqui é agrupar os salários por departamento. Pois bem, como pode ser visto na ilustração, neste caso, não conseguimos montar o agrupamento. Note que na coluna de departamento é possível agrupar os valores, mas na coluna de empregados isso não é possível.

Como a coluna de empregados faz parte do `SELECT`, ela acaba comprometendo todo nosso agrupamento. Atenção a um detalhe – o fato de a coluna `Empregado` estar sendo visualizada primeiro não quer dizer que seja a causa de não conseguirmos agrupar por departamento. A ordem das colunas não altera o resultado. Vamos retirá-la do nosso `SELECT`.

Empregado
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK

Departamento	Salário
RESEARCH	800
SALES	1600
SALES	1250
RESEARCH	2975
SALES	1250
SALES	2850
ACCOUNTING	2450

Fig. 10.3

Agora sim. Tiramos a coluna de empregados e ficamos apenas com as colunas `Departamento` e `Salários`.

Veja como ficou nosso agrupamento:

Agrupamento por Group By

Agrupamento por somatória

Departamento	Salário
RESEARCH	800
RESEARCH	2975
SALES	1600
SALES	1250
SALES	1250
SALES	2850
ACCOUNTING	2450

Resultado final: Valores agrupados por departamento

Fig. 10.4

Fazendo dessa forma conseguimos alcançar nosso objetivo.

Resumindo

- Devemos saber que, para obter sucesso em nossos agrupamentos, as colunas que não estão sendo agrupadas pelas funções de agrupamento devem ser agrupadas pelo `GROUP BY`;
- Também podemos agrupar determinadas colunas, mesmo que elas não estejam presentes na cláusula `SELECT`;
- Somente vamos precisar agrupar colunas através do `GROUP BY` quando desejarmos mostrar um resultado com base em outro. Por

exemplo, valores de salário por departamento, quantidades de empregados por departamento e assim por diante. Se quisermos apenas saber a somatória de todos os valores de salário, independente do departamento ou de qualquer outra informação, não precisamos utilizar o `GROUPY BY`;

- Funções de agregação, no geral, ignoram valores nulos;
- Para realizar o agrupamento de informações, o Oracle poderá ordenar ou não as colunas. Caso a coluna que está sob a ação da função for uma coluna com índice, o banco poderá utilizá-lo. Como os índices são ordenados, não será necessário ordenar os dados para o agrupamento. Caso contrário, ele vai realizar a ordenação primeiro, e depois, agrupar. Nem todas as funções permitem usar índices.

Agora vamos ver esses conceitos na prática.

Primeiramente, visualizamos o nome de todos os empregados, os nomes dos seus departamentos e seus respectivos salários.

```
SQL> select ename, dname, sal
     from emp e, dept d
     where e.deptno = d.deptno
     order by ename;

ENAME      DNAME                SAL
---------- -------------- ---------
ADAMS      RESEARCH            1100
ALLEN      SALES               1600
BLAKE      SALES               2850
CLARK      ACCOUNTING          2450
FORD       RESEARCH            3000
JAMES      SALES                950
JOHN       RESEARCH            1000
JONES      RESEARCH            2975
KING       ACCOUNTING          5000
MARTIN     SALES               1250
MILLER     ACCOUNTING          1300
SCOTT      RESEARCH            3000
```

```
SMITH      RESEARCH            800
TURNER     SALES              1500
WARD       SALES              1250

15 rows selected.

SQL>
```

Através de um comando SQL selecionamos os mesmos dados do `SELECT` anterior, mas agora sumarizando os salários. Note que continuamos selecionando as colunas do nome do empregado e do departamento do empregado. O objetivo do programa é mostrar a soma dos salários por departamento.

```
SQL> select ename, dname, sum(sal)
     from emp e, dept d
     where e.deptno = d.deptno
     order by ename;
select ename, dname, sum(sal)
       *
ERROR at line 1:
ORA-00937: not a single-group group function


SQL>
```

Ao executar o programa, surgiu um erro que, em linhas gerais, quer nos dizer que o comando `SELECT` está tentando utilizar uma função de grupo, juntamente com outras colunas não agrupadas, sem a cláusula de agrupamento. Como visto nos conceitos apresentados anteriormente, isso não é permitido. Devemos agrupar as colunas que não estão associadas a funções de agrupamento. Veja a seguir como ficou.

```
SQL> select ename, dname, sum(sal)
     from emp e, dept d
     where e.deptno = d.deptno
     group by ename, dname
     order by ename;
ENAME      DNAME            SUM(SAL)
---------- -------------- ---------
```

```
ADAMS       RESEARCH             1100
ALLEN       SALES                1600
BLAKE       SALES                2850
CLARK       ACCOUNTING           2450
FORD        RESEARCH             3000
JAMES       SALES                 950
JOHN        RESEARCH             1000
JONES       RESEARCH             2975
KING        ACCOUNTING           5000
MARTIN      SALES                1250
MILLER      ACCOUNTING           1300
SCOTT       RESEARCH             3000
SMITH       RESEARCH              800
TURNER      SALES                1500
WARD        SALES                1250

15 rows selected.
SQL>
```

Feitos os agrupamentos necessários, voltamos a executar o `SELECT`. O resultado foi apresentado logo em seguida. No entanto, veja que algo não saiu como deveria. Os salários não foram sumarizados por departamento e, sim, por empregado. Desta forma, seria a mesma coisa que não sumarizar. Vamos alterar o comando retirando a coluna nome do empregado do comando SQL.

```
SQL> select dname, sum(sal)
     from emp e, dept d
     where e.deptno = d.deptno
     group by ename, dname
     order by ename;
DNAME            SUM(SAL)
-------------- ---------
RESEARCH             1100
SALES                1600
SALES                2850
ACCOUNTING           2450
RESEARCH             3000
SALES                 950
RESEARCH             1000
```

```
RESEARCH            2975
ACCOUNTING          5000
SALES               1250
ACCOUNTING          1300
RESEARCH            3000
RESEARCH             800
SALES               1500
SALES               1250

15 rows selected.
SQL>
```

Ao retirar a coluna, o erro persiste. Isso acontece pois não adianta retirar apenas da seleção, mas também é necessário retirar do agrupamento. Repare que na linha 4 ainda consta a coluna `ENAME`. Veja a seguir como deve ficar o `SELECT`, para que se consiga atingir o objetivo proposto.

```
SQL> select dname, sum(sal)
     from emp e, dept d
     where e.deptno = d.deptno
     group by dname;

DNAME            SUM(SAL)
-------------- ---------
ACCOUNTING          8750
RESEARCH           11875
SALES               9400

SQL>
```

Selecionando apenas a coluna referente ao nome do departamento e sumarizando os salários, através da função de agregação `SUM`, temos como resultado a soma dos salários por departamento.

Seguem as funções de agregação mais utilizadas:

- **COUNT:** retorna a quantidade de incidências de registros;
- **SUM:** exibe a soma dos valores dos registros;

- **AVG:** exibe a média dos valores de uma determinada coluna;
- **MIN:** exibe o menor valor de uma coluna;
- **MAX:** retorna o maior valor de uma coluna.

O primeiro exemplo mostra o uso da função COUNT. O objetivo é selecionar a quantidade de empregados por país.

```
SQL> select count(employee_id) cont_emp, country_name
     from   employees e
           ,departments d
           ,locations l
           ,countries c
     where e.department_id = d.department_id
     and d.location_id = l.location_id
     and l.country_id = c.country_id
     group  by country_name
     order  by country_name;

 CONT_EMP COUNTRY_NAME
--------- ----------------------------------------
        2 Canada
        1 Germany
       35 United Kingdom
       68 United States of America

SQL>
```

O próximo exemplo é quase igual ao anterior, apenas por um detalhe. Note que, como conhecemos a tabela EMPLOYEES e sabemos que existe apenas um registro para cada empregado, podemos usar a função COUNT de outra forma, utilizando asterisco (*) no lugar da coluna EMPLOYEE_ID.

```
SQL> select count(*) cont_emp, country_name
     from   employees e
           ,departments d
           ,locations l
           ,countries c
```

```
     where e.department_id = d.department_id
     and d.location_id = l.location_id
     and l.country_id = c.country_id
     group  by country_name
     order  by country_name;

 CONT_EMP COUNTRY_NAME
--------- ----------------------------------------
        2 Canada
        1 Germany
       35 United Kingdom
       68 United States of America

SQL>
```

O próximo exemplo mostra o uso da função `MAX`. O objetivo é selecionar a maior data de Admissão de cada departamento.

```
SQL> select dname, max(hiredate)
     from emp e, dept d
     where e.deptno = d.deptno
     group by dname
     order by 2 desc;

DNAME          MAX(HIRED
-------------- ---------
RESEARCH       30-MAR-11
ACCOUNTING     23-JAN-82
SALES          03-DEC-81

SQL>
```

Este exemplo mostra o uso da função `MIN`. O objetivo é selecionar a menor data de admissão entre todos os empregados:

```
SQL> select min(hire_date) from employees;

MIN(HIRE_
---------
```

```
17-JUN-87
SQL>
```

Além das funções de agregação e do uso do `GROUP BY`, também podemos contar com o `HAVING` para nos ajudar a restringir registros com base nos valores retornados pelas funções de agregação.O `HAVING` existe pois não podemos utilizar funções de agregação na cláusula `WHERE`.

No exemplo a seguir, é mostrado o uso da cláusula `HAVING`. O objetivo é selecionar a quantidade de empregados e a soma dos salários, agrupados por departamentos cuja quantidade de empregados é maior que 5. Observe que a cláusula `HAVING` atua somente após o agrupamento das linhas. Por isso, não seria possível utilizar a cláusula `WHERE`, já que ela atua no momento em que as linhas estão sendo selecionadas, ou seja, antes do agrupamento.

```
SQL> select count(employee_id) cont_emp,
     sum(salary) soma_salario, department_name
     from   employees e
           ,departments d
     where e.department_id = d.department_id
     having count(employee_id) > 5
     group  by department_name
     order  by department_name;

 CONT_EMP SOMA_SALARIO DEPARTMENT_NAME
--------- ------------ ------------------------------
        6        51600 Finance
        6        24900 Purchasing
       34       304500 Sales
       45       156400 Shipping
SQL>
```

Já neste outro exemplo, o objetivo é selecionar a soma dos salários, agrupados por departamento e país, sendo que a soma é maior que a média dos salários por país.

```
SQL> select department_name, sum(salary), country_name
     from   employees e
           ,departments d
```

```
           ,locations l
           ,countries c
     where e.department_id = d.department_id
     and d.location_id = l.location_id
     and l.country_id = c.country_id
     having sum(salary) > (select avg(em.salary)
                           from   employees em
                                 ,departments dm
                                 ,locations lm
                                 ,countries cm
                           where  em.department_id =
                                         dm.department_id
                           and    dm.location_id =
                                         lm.location_id
                           and    lm.country_id = cm.country_id
                           and    cm.country_id = c.country_id)
     group by c.country_id
             ,department_name
             ,country_name
     order by country_name
             ,department_name;

DEPARTMENT_NAME              SUM(SALARY) COUNTRY_NAME
---------------------------- ----------- ------------------------
Marketing                          19000 Canada
Sales                             304500 United Kingdom
Accounting                         20300 United States of America
Executive                          58000 United States of America
Finance                            51600 United States of America
IT                                 28800 United States of America
Purchasing                         24900 United States of America
Shipping                          156400 United States of America

8 rows selected.
SQL>
```

10.3 Funções de data

Funções de data são utilizadas para manipular valores do tipo `date`, como aplicar formatações para uma visualização mais refinada, ou extrair partes de uma data, as horas, dia do mês ou somente o ano. Seguem algumas destas funções:

- **ADD_MONTHS:** adiciona meses em uma determinada data;
- **MONTHS_BETWEEN:** retorna a quantidade de meses entre duas datas;
- **NEXT_DAY:** procura o próximo dia após uma data informada;
- **LAST_DAY:** retorna o último dia do mês com base em uma data informada;
- **TRUNC:** trunca uma data passada por parâmetro. Pode ser feito por dia e mês, utilizando o parâmetro `FMT` (formato);
- **SYSDATE:** retorna a data corrente com base no servidor do banco de dados;
- **SESSIONTIMEZONE:** mostra o fuso horário com base na sessão aberta no banco de dados, mediante sua localização. Vale lembrar que os fusos horários são calculados com base no meridiano de Greenwich.
- **CURRENT_DATE:** mostra a data corrente com base na zona de tempo da sessão do usuário. A Zona de tempo é afetada em relação ao Meridiano. Caso não haja mudanças de zona, esta função terá o mesmo valor que `SYSDATE`, que busca a hora do servidor do banco de dados, mesmo que a sessão tenha sido aberta em uma zona diferente daquela em que o servidor se encontra. Já o `CURRENT_DATE` refletirá a zona onde foi aberta a sessão.

Quando for calcular a quantidade de anos entre duas datas, utilize a função MONTHS_BETWEEN, dividindo o retorno da função (quantidade de meses entre as datas) por 12 (quantidade de meses em um ano). Desta forma, você chegará a um valor mais preciso do que subtraindo duas datas e dividindo (retorno em dias) por 365 dias. O resultado final, em ambas as formas, pode ser truncado.

Seguem alguns exemplos. No primeiro, é mostrado o uso da função NEXT_DAY. O exemplo seleciona os empregados que têm como mês de aniversário o mês corrente. Através dessa função, é calculado qual o primeiro domingo logo em seguida ao dia de aniversário de cada empregado.

```
SQL> select first_name
          ,to_date(to_char(birth_date,'dd/mm')||'/'||
                     to_char(sysdate,'rrrr')
          ,'dd/mm/rrrr') dt_aniversario
          ,next_day(to_date(to_char(birth_date,'dd/mm')||'/'||
                     to_char(sysdate,'rrrr'),'dd/mm/rrrr')
                               ,'SUNDAY') DOMINGO
     from   employees
     where to_char(birth_date,'mm') = to_char(sysdate,'mm')
     order  by 2;
FIRST_NAME           DT_ANIVER DOMINGO
-------------------- --------- ---------
Joshua               06-APR-11 10-APR-11
Adam                 10-APR-11 17-APR-11
TJ                   10-APR-11 17-APR-11
Amit                 21-APR-11 24-APR-11
Sundita              21-APR-11 24-APR-11
Jack                 23-APR-11 24-APR-11
Alana                24-APR-11 01-MAY-11

7 rows selected.
SQL>
```

A seguir, é mostrado o uso das funções ADD_MONTHS e MONTHS_BETWEEN. Selecionamos os empregados e suas respectivas da-

tas de término de experiência do cargo. Note que limitamos o número de empregados no SELECT através da função ADD_MONTHS, para evitar o retorno de todas as linhas da tabela.

```
SQL> select ename, dname
           ,hiredate
           ,add_months(hiredate,3) dt_termino_exp
           ,to_char(months_between(sysdate,hiredate),'90D00')
              qt_meses_trabalho
     from emp e, dept d
     where e.deptno = d.deptno
     and add_months(hiredate,3) >= sysdate;

ENAME      DNAME          HIREDATE  DT_TERMIN QT_MES
---------- -------------- --------- --------- ------
JOHN       RESEARCH       30-MAR-11 30-JUN-11   0.14
SQL>
```

No exemplo seguinte, é mostrado o uso das funções SESSIONTIMEZONE, CURRENT_DATE e SYSDATE. São exibidos o fuso horário, a data corrente local e data atual do servidor.

```
SQL> select sessiontimezone
           ,current_date
           ,sysdate
     from   dual;
SESSIONTIMEZONE                             CURRENT_D SYSDATE
------------------------------------------- --------- ---------
-03:00                                      03-APR-11 03-APR-11

SQL>
```

O próximo exemplo mostra o uso das funções LAST_DAY, ROUND e TRUNC. São apresentadas várias formas de usar as funções para extrair ou manipular determinadas informações, como, por exemplo, arredondar e truncar datas.

```
SQL> select ename
           ,hiredate
```

```
        ,last_day(hiredate) last
        ,round(hiredate, 'YEAR') round
        ,trunc(hiredate, 'YEAR') trunc
    from emp;

ENAME      HIREDATE  LAST      ROUND     TRUNC
---------- --------- --------- --------- ---------
SMITH      17-DEC-80 31-DEC-80 01-JAN-81 01-JAN-80
ALLEN      20-FEB-81 28-FEB-81 01-JAN-81 01-JAN-81
WARD       22-FEB-81 28-FEB-81 01-JAN-81 01-JAN-81
JONES      02-APR-81 30-APR-81 01-JAN-81 01-JAN-81
MARTIN     28-SEP-81 30-SEP-81 01-JAN-82 01-JAN-81
BLAKE      01-MAY-81 31-MAY-81 01-JAN-81 01-JAN-81
CLARK      09-JUN-81 30-JUN-81 01-JAN-81 01-JAN-81
SCOTT      09-DEC-82 31-DEC-82 01-JAN-83 01-JAN-82
KING       17-NOV-81 30-NOV-81 01-JAN-82 01-JAN-81
TURNER     08-SEP-81 30-SEP-81 01-JAN-82 01-JAN-81
ADAMS      12-JAN-83 31-JAN-83 01-JAN-83 01-JAN-83
JAMES      03-DEC-81 31-DEC-81 01-JAN-82 01-JAN-81
FORD       03-DEC-81 31-DEC-81 01-JAN-82 01-JAN-81
MILLER     23-JAN-82 31-JAN-82 01-JAN-82 01-JAN-82
JOHN       30-MAR-11 31-MAR-11 01-JAN-11 01-JAN-11

15 rows selected.

SQL>
```

10.4 Funções de conversão

Em muitos casos, precisamos converter um determinado dado de um tipo para outro. O Oracle disponibiliza funções de conversão para este trabalho. Estas funções, embora sejam simples de serem usadas, ajudam muito no momento de converter ou formatar dados provenientes dos seus comandos SQL.

- **TO_DATE:** converte uma string (`char` ou `varchar2`) de caractere para uma data;
- **TO_NUMBER:** converte uma string (`char` ou `varchar2`) de caractere para um número;

- **TO_CHAR:** converte um número ou uma data para uma string de caractere.

Cada função possui suas características e são utilizadas para cumprir um objetivo diferente. O que elas possuem em comum é o número de parâmetros. O primeiro parâmetro está relacionado ao valor que deve ser convertido; o segundo corresponde ao formato que você deseja aplicar; e por último e opcional, o parâmetro de linguagem. Trata-se de funções muito utilizadas no dia a dia, e é de fundamental importância, conhecê-las e entender como se comportam. Além das mencionadas, a Oracle disponibiliza várias outras para as mais diversas situações. Contudo, essas três são as que mais comumente utilizamos nas construções de nossos comandos SQL. Para saber mais sobre outras funções de conversão, consulte em livros de Oracle que falam sobre a SQL ou na documentação disponível no site da Oracle.

TO_DATE

Esta função é bem interessante. Com um pouco de treino e criatividade, podemos realizar várias conversões que podem ajudar muito no momento de formatar, selecionar ou criticar os dados retornados de um comando SQL.

Ela funciona basicamente da seguinte forma: você vai passar um valor caractere para a função, juntamente com um formato. Este formato deve ser compatível com o conjunto de caracteres que você passou para a função. A qualquer incompatibilidade, o Oracle gera um erro de conversão.

Exemplo: `TO_DATE('21/05/2009','dd/mm')`. Esta conversão vai gerar um erro, pois você está informando dia, mês e ano, como caractere, mas na máscara só mencionou dia e mês. Quando o Oracle vai fazer a conversão, ele analisa o formato que você está passando como parâmetro e verifica o que é elemento de função e o que é caractere. Neste caso, ele sabe que `dd` e `mm` são elementos conhecidos de dia e mês, e que `/` é um caractere que serve como uma espécie de separador. Então, depois desta identificação, ele pega cada caractere informado e vai convertendo conforme o formato `2=d`, `1=d`, `/=/`, `0=m`, `5=m` etc. Mas quando ele chega na segunda `/`, vê que não há formato para o caractere, pois terminou no elemento `m`. Logo, é gerado um erro de conversão.

Veja a execução:

```
SQL> select TO_DATE('21/05/2009','dd/mm') from dual;
select TO_DATE('21/05/2009','dd/mm') from dual
               *
ERROR at line 1:
ORA-01830: date format picture ends before converting entire
input string


SQL>
```

O contrário também gera outro erro:

```
SQL> select TO_DATE('21/05','dd/mm/yyyy') from dual;
select TO_DATE('21/05','dd/mm/yyyy') from dual
               *
ERROR at line 1:
ORA-01840: input value not long enough for date format


SQL>
```

Seguem exemplos do uso do `TO_DATE`. O primeiro mostra como é possível converter strings em datas válidas. Veja que, na linha 4 do programa, temos a função convertendo a string `010182` para uma data utilizando o formato DDMMRR.

```
SQL> select ename
           ,hiredate
     from emp
     where hiredate > to_date('010182','ddmmrr');

ENAME      HIREDATE
---------- ---------
SCOTT      09-DEC-82
ADAMS      12-JAN-83
MILLER     23-JAN-82
JOHN       30-MAR-11

SQL>
```

Já neste outro exemplo, também utilizando `TO_DATE` e `SELECT`, temos a função convertendo outra string em uma data, utilizando um formato diferente.

```
SQL> select to_date('21.05.2009','dd.mm.yyyy') from dual;

TO_DATE('
---------
21-MAY-09

SQL>
```

Podemos também adicionar à chamada da função aspectos referentes à linguagem. Neste exemplo, estamos convertendo uma string por extenso em data, utilizando o formato americano:

```
SQL> select to_date('April 21','month dd',
     'nls_date_language=american') from dual;

TO_DATE('
---------
21-APR-11

SQL>
```

Contudo, como pode ser visto no exemplo anterior, a visualização continua sendo no formato americano, embora estejamos convertendo a string para data utilizando o formato brasileiro. Lembre-se, conversão não necessariamente tem a ver com a forma com que o dado será impresso na tela.

```
SQL> select to_date('Abril 21','month dd',
'nls_date_language=''BRAZILIAN PORTUGUESE''') from dual;

TO_DATE('
---------
21-APR-11

SQL>
```

Este exemplo mostra que devemos informar formatos válidos, ou melhor, formatos conhecidos da linguagem. Caso contrário, a conversão não é realizada e erros ocorrerão:

```
SQL> select to_date('Abril 21','month XX',
'nls_date_language=''BRAZILIAN PORTUGUESE''') from dual;
select to_date('Abril 21','month XX',
'nls_date_language=''BRAZILIAN PORTUGUESE''') from dual
                                *
ERROR at line 1:
ORA-01821: date format not recognized


SQL>
```

A seguir, algumas limitações com relação à função `TO_DATE`:

- A string a ser passada para a conversão não pode conter mais de 220 caracteres;
- Existem vários formatos de máscara disponíveis para a utilização. Qualquer máscara diferente das permitidas pela Oracle gerará um erro de conversão;
- Não pode haver confronto de máscaras. Por exemplo, você queria utilizar a máscara `HH24` e também solicitar que fosse mostrado `AM` (indicador de antemeridiano para manhã) ou `PM` (indicador de pós-meridiano para noite);
- Não é permitido especificar elementos de conversão duplicados. Exemplo: `DD-MM-MM`. Neste caso, o formato para mês aparece duas vezes.

Veja alguns elementos de formatação que podem ser usados:

- **CC:** adiciona 1 aos dois primeiros dígitos do ano (YYYY);
- **SCC:** igual CC, prefixando datas BC com um sinal negativo;
- **YY:** representa o ano com duas casas;

- **YYYY:** representa o ano com quatro casas;
- **RR:** representa os dois últimos dígitos do ano, mas obedecendo a seguinte regra: soma 1 aos dois primeiros dígitos de CC se ano for < 50 e os últimos 2 dígitos do ano corrente forem >= 50. Subtrai 1 de CC se ano >= 50 e os últimos dois dígitos do ano corrente forem < 50;
- **RRRR:** representa o ano. Aceita 2 ou 4 dígitos. Se ano informado com 2 dígitos, segue as mesmas regras de `RR`;
- **YEAR:** escreve o ano por extenso;
- **MM:** número do mês de 01 a 12. 01 = Janeiro, 02 = Fevereiro etc.;
- **MONTH:** nome do mês;
- **MON:** representa o nome do mês abreviado com três caracteres;
- **DD:** dia do mês de 1 a 31;
- **DDD:** representa o dia do ano de 1 a 366;
- **DAY:** representa o nome do dia por extenso;
- **HH, HH12, HH24:** HH e HH12, horas de 1 a 12. HH24, horas de 0 a 23;
- **MI:** equivale aos minutos de 0 a 59;
- **SS:** equivale aos segundos de 0 a 59;
- **SP:** converte o número para seu formato escrito. Disponível apenas para a escrita no idioma inglês;
- **SPTH:** mostra os números de maneira ordinal. 1 = First, 2 = Second etc.;
- **FM:** retira espaços em branco proveniente da ausência de caracteres em um formato.

Esses elementos também são utilizados na conversão do tipo `DATE` para string.

A função `TO_DATE` pode converter não só strings representando datas completas, como também strings representando partes de uma data. Este exemplo mostra a função realizando a conversão da string `2008` em data. Note que, ao ser convertida, recebeu a data atual, modificada apenas pelo ano convertido:

```
SQL> select to_date('2008','yyyy') from dual;

TO_DATE('
---------
01-APR-08

SQL>
```

Este exemplo é similar ao anterior. Neste, está sendo realizada a conversão com base na representação numérica do dia referente ao total de dias do ano:

```
SQL> select to_date(200,'ddd') from dual;

TO_DATE(2
---------
19-JUL-11

SQL>
```

Uso de YYYY e RRRR

Sabemos que a máscara `YYYY` representa os quatro dígitos do ano. Opcionalmente, pode-se utilizar `YY` para mostrar apenas os dois últimos dígitos. Contudo, para corrigir problemas de compatibilidade com a virada do século, a Oracle criou as máscaras `RR` e `RRRR`. Veja a aplicação a seguir:

```
SQL> select to_char(sysdate,'dd/mm/yyyy') from dual;
```

```
TO_CHAR(SY
----------
14/04/2011

SQL> select to_char(to_date('01/01/49','dd/mm/yy'),'dd/mm/yyyy')
     from dual;

TO_CHAR(TO
----------
01/01/2049

SQL> select to_char(to_date('01/01/50','dd/mm/yy'),'dd/mm/yyyy')
     from dual;

TO_CHAR(TO
----------
01/01/2050

SQL> select to_char(to_date('01/01/49','dd/mm/rr'),'dd/mm/rrrr')
     from dual;

TO_CHAR(TO
----------
01/01/2049

SQL> select to_char(to_date('01/01/50','dd/mm/rr'),'dd/mm/rrrr')
     from dual;

TO_CHAR(TO
----------
01/01/1950

SQL>
```

Nesse exemplo, vemos formatações de datas utilizando a função `TO_CHAR`, juntamente com os elementos de formatação `YY` e `RR`. Note que podemos ter datas diferentes ao converter utilizando `RR` e `YY`, dependendo do ano da data informada.

Para mais detalhes e compreensão desse exemplo, consulte as regras do uso do formato RR na lista de elementos de formatação vista anteriormente.

TO_NUMBER

Muito semelhante à função TO_DATE, esta função também tem o papel de converter determinados valores. Seu objetivo, no entanto, é fazer a conversão de caracteres para numéricos.

Quando falamos na função TO_DATE, foi mencionado que o valor do caractere que está sendo informado como parâmetro deve ser compatível com o formato. Pois bem, quando trabalhamos com TO_NUMBER, o mesmo também acontece. Ao informarmos um valor caractere para a função TO_NUMBER, ele deve ser compatível com o formato que estamos passando para a função. Todavia, existe uma particularidade quanto ao formato para casas decimais e de grupo (milhar, por exemplo).

Para cálculos internos do Oracle, sempre será usado ponto (.) como separador decimal, e vírgula (,) para separador de grupo, como padrão americano. Para atribuições de variáveis do tipo caractere ou para visualização, o Oracle pegará a formatação conforme estiver configurado na variável NLS_NUMERIC_CHARACTERS (decimal, milhar).

Uma das razões pelas quais o Oracle utiliza essa premissa pode ser pelo fato de a vírgula ser utilizada para a separação de valores ou colunas em um comando SQL. Veja o exemplo, no qual fazemos um SELECT do número 111,1 utilizando vírgula como separador decimal:

```
SQL> select 111,1 from dual;

       111          1
---------- ----------
       111          1

SQL>
```

Agora, executando o mesmo SELECT, mas utilizando o ponto como separador decimal:

```
SQL> select 111.1 from dual;
```

```
     111.1
----------
     111.1

SQL>
```

No primeiro exemplo, o comando SQL acabou entendendo que `111` era um dado e `1` era outro. O mesmo não aconteceu quando, em vez de separarmos por vírgula, separamos por ponto.

Vejamos o exemplo a seguir. Nele, está se tentando converter um valor no qual temos como separador de decimais a vírgula, e para milhar, o ponto. Pois bem, já sabemos que para cálculos de conversões internas o Oracle utiliza o ponto como decimal. Logo, o comando SQL a seguir gera um erro:

```
SQL> select to_number('4.569.900,87') valor
     from   dual;
select to_number('4.569.900,87') valor
                 *
ERROR at line 1:
ORA-01722: invalid number


SQL>
```

Ok! O erro acontece porque, onde há vírgula, deveria ter ponto, e onde há ponto, deveria ter vírgula. Pois bem, vejamos o `SELECT` a seguir:

```
SQL> select to_number('4,569,900.87') valor
     from   dual;
select to_number('4,569,900.87') valor
                 *
ERROR at line 1:
ORA-01722: invalid number


SQL>
```

Agora você deve estar se perguntando: mas por que o erro, sendo que tudo indica que agora o formato foi informado corretamente, com ponto para decimais e vírgula para milhares? Veja o `SELECT` a seguir:

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300
     8000 JOHN       CLERK           7902 30-MAR-11       1000

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10
      200        20

15 rows selected.
```

```
SQL>
```

Observe a coluna SAL, mais precisamente o registro EMPNO=7566. Veja que o valor do salário está aparecendo como 3272.5. Este número possui como separador de decimais o caractere ponto, mas não possui separador de grupo de milhar.

Isso nos leva a conclusão de que, além de sabermos as regras para pontos e vírgulas, também devemos estar cientes de como está definida a sessão do Oracle. No Oracle, a formatação de milhar não aparece, a menos que seja aplicada a formatação para tal. Vamos voltar ao primeiro exemplo, agora aplicando uma formatação:

```
SQL> select to_number('4.569.900,87','9G999G999D00') valor
     from   dual;
select to_number('4.569.900,87','9G999G999D00') valor
                 *
ERROR at line 1:
ORA-01722: invalid number


SQL>
```

Ainda assim, o erro persiste. Também pudera, vimos no registro EMPNO=7566 que na sessão do Oracle está definido como casa decimal o ponto, e não a vírgula. Vamos trocar:

```
SQL> select to_number('4,569,900.87','9G999G999D00') valor
     from   dual;

    VALOR
---------
4569900.9

SQL>
```

Bingo! Este comando SELECT funcionou por duas razões. Primeira: o valor passado no formato caractere possui um número válido que contém

como separador de casas decimais o mesmo definido na sessão do Oracle. Segundo: foi definido um formato para o qual estipulamos que, neste conjunto de caracteres, há separadores de grupo que representamos com o elemento de função `G`. Utilizando desta forma, o Oracle entende que o ponto é o separador de decimal e a vírgula, o de milhar. Para provar nossa tese, vamos mudar os separadores na sessão do Oracle e executar o mesmo `SELECT` anterior:

```
SQL> alter session set nls_numeric_characters='.,';

Session altered.

SQL> select to_number('4.569.900,87','9G999G999D00') valor
     from   dual;
select to_number('4.569.900,87','9G999G999D00') valor
                 *
ERROR at line 1:
ORA-01722: invalid number


SQL>
```

Agora vamos trocar os caracteres de lugar:

```
SQL> select to_number('4,569,900.87','9G999G999D00') valor
     from   dual;

    VALOR
---------
4569900.9
SQL>
```

Funcionou. Mas espere um momento! Além de converter, preciso saber como está definido na sessão do Oracle? Não. Não é necessário se você especificar, além do formato, quais caracteres devem ser utilizados para a separação de decimais e grupos. Vejamos novamente o exemplo anterior ao último. Nele, definimos que na sessão seria definido que o ponto seria a decimal, e vírgula, o grupo. Contudo, isto gerou um erro ao executar nosso `SELECT` onde estava justamente definido ao contrário da parametrização feita.

```
SQL> alter session set nls_numeric_characters='.,';

Session altered.

SQL> select to_number('4.569.900,87','9G999G999D00') valor
     from   dual;
select to_number('4.569.900,87','9G999G999D00') valor
                 *
ERROR at line 1:
ORA-01722: invalid number
SQL>
```

Utilizando o parâmetro `NLS`, aquele terceiro parâmetro da função que é opcional, nós conseguimos, em vez de alterar a sessão do Oracle, validar esses caracteres apenas no nosso comando. Vamos executar novamente o comando `SELECT` anterior, mas agora incorporando o terceiro parâmetro:

```
SQL> select to_number('4.569.900,87','9G999G999D00',
     'nls_numeric_characters=,.') valor
     from   dual;

    VALOR
---------
4569900.9

SQL>
```

Funcionou perfeitamente. Embora pareça complicado no início, você vai se acostumando com as características e logo pega a lógica. Vale salientar que esta questão da sessão também é válida para o `TO_DATE`.

Além da utilização dos elementos de formatação `G` (grupo) e `D` (decimal), você pode optar por utilizar em seu formato o ponto `.` e a vírgula `,`. Veja o exemplo:

```
SQL> select to_number('4,569,900.87','9,999,999.00') valor
     from   dual;
     VALOR
----------
4569900,87
```

```
SQL>
```

Contudo, mesmo informando elementos fixos, por assim dizer, temos que respeitar a definição padrão do Oracle para conversões internas (chamadas, implícitas). Dessa forma, se tentarmos impor um formato para o Oracle, por exemplo, utilizando vírgula para decimais, mesmo sendo de forma fixa, ele não permitirá. Veja a seguir:

```
SQL> select to_number('4.569.900,87','9.999.999,00') valor
     from   dual;
select to_number('4.569.900,87','9.999.999,00') valor
*
ERRO na linha 1:
ORA-01481: modelo inválido para formato de número
```

Observe que, mesmo informando a máscara de acordo com o número informado, o Oracle não aceitou a conversão. Ele desconhece o formato fixo, com decimais sendo separados por vírgula.

Mas se tentarmos modificar o padrão, alterando assim a sessão? Vai funcionar?

```
SQL> alter session set nls_numeric_characters=',.';
Sessão alterada.
SQL> select to_number('4.569.900,87','9.999.999,00') valor
     from   dual;
select to_number('4.569.900,87','9.999.999,00') valor
*
ERRO na linha 1:
ORA-01481: modelo inválido para formato de número
SQL>
```

Veja que o erro permanece, pois essa alteração reflete apenas para o uso dos elementos `G` e `D`, e não para o uso fixo de elementos. Neste caso, para conseguirmos converter o valor, utilizando vírgula como separador decimal, temos que usar os elementos de formato, mesmo que já tenhamos alterado a sessão para tal. Agora, sim, a forma correta:

```
SQL> alter session set nls_numeric_characters=',.';
Sessão alterada.

SQL> select to_number('4.569.900,87','9G999G999D00') valor
     from   dual;
     VALOR
----------
4569900,87

SQL>
```

Formatos de sessão para datas

Abrindo um parênteses dentro deste tópico, salientamos que, além das alterações referente aos separadores numéricos, também podemos alterar as sessões para definir um novo formato de data e também para a definição de linguagens. Veja exemplos de definição de formato para a sessão:

```
alter session set nls_language      = 'BRAZILIAN PORTUGUESE';
alter session set nls_date_language = 'PORTUGUESE';
alter session set nls_date_format   = 'DD/MM/RRRR';
```

Agora, na prática, veja o `SELECT`, observando a coluna `HIREDATE`:

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
```

```
      7900 JAMES      CLERK          7698 03-DEC-81       950
      7902 FORD       ANALYST        7566 03-DEC-81      3000
      7934 MILLER     CLERK          7782 23-JAN-82      1300
      8000 JOHN       CLERK          7902 30-MAR-11      1000

      COMM    DEPTNO
--------- ---------
                 20
       300        30
       500        30
                 20
      1400        30
                 30
                 10
                 20
                 10
         0        30
                 20
                 30
                 20
                 10
       200        20

15 rows selected.

SQL>
```

Para o registro `EMPNO=7566`, o valor da coluna `HIREDATE` é `02-APR-81`. Logo, este é o formato definido na sessão. Vamos olhar o próximo `SELECT`:

```
SQL> select to_date('02-APR-81') from dual;

TO_DATE('
---------
02-APR-81

SQL>
```

Nota-se que não foi gerado erro, mesmo não sendo informado o formato. Isso aconteceu porque o valor de caractere informado como string está no mesmo formato da sessão do Oracle. Veja o que acontece quando colocamos a string em outro formato:

```
SQL> select to_date('05/21/2009') from dual;
select to_date('05/21/2009') from dual
               *
ERROR at line 1:
ORA-01843: not a valid month


SQL>
```

Para consertar isso, devemos informar um formato compatível com o conjunto de caracteres informado ou trocar o formato da sessão do Oracle.

Primeiro veja a solução informando um formato compatível:

```
SQL> select to_date('05/21/2009','mm/dd/yyyy') from dual;

TO_DATE('
---------
21-MAY-09

SQL>
```

Agora, alterando a sessão do Oracle:

```
SQL> alter session set nls_date_format = 'mm/dd/yyyy';

Session altered.

SQL> select to_date('05/21/2009') from dual;

TO_DATE('0
----------
05/21/2009

SQL>
```

Veja alguns elementos de formatação que podem ser usados:

- **9:** cada nove representa um caractere que será substituído pelo caractere referente ao valor numérico passado como parâmetro. Os zeros na frente são tratados como espaços em branco. **Exemplo:** 999999 **Resutado:**234
- **0:** adicionando 0 como um prefixo ou sufixo ao número, todos os zeros iniciais ou finais são tratados e exibidos como zeros em vez de um espaço em branco. **Exemplo:** 099999 **Resutado:**001234
- **$:** prefixo do símbolo de moeda impresso na primeira posição. **Exemplo:** $999999 **Resutado:**$1234
- **S:** exibe um sinal de + inicial ou final quando o valor for positivo, e um sinal de – inicial ou final quando o valor for negativo;
- **D:** localização do ponto decimal. Os noves de ambos os lados refletem o número máximo de dígitos permitidos. **Exemplo:** 99D99 **Resutado:**99.99
- **G:** especifica um separador de grupo (milhar, por exemplo) como uma vírgula. **Exemplo:** 9G999 **Resutado:**9,999
- **L:** especifica a localização do símbolo de moeda local (tal como $). **Exemplo:** L999999 **Resutado:**FF1234
- **,:** coloca uma vírgula na posição especificada, independentemente do separador de grupo. **Exemplo:** 999,999 **Resutado:**999,999 1,234
- **.:** especifica a localização do ponto decimal, independentemente do separador decimal. **Exemplo:** 999999.99 **Resutado:**1234.00
- **FM:** remove os espaços em branco inicial e final. **Exemplo:** FM9999 **Resutado:**9999
- **MI:** Sinal de subtração à direita (valores negativos). **Exemplo:** 999999MI **Resultado:** 1234-

- **PR:** Números negativos entre colchetes. **Exemplo:** 999999PR **Resultado:** <1234>
- **EEEE:** Notação científica (o formato deve especificar quatro Es). **Exemplo:** Notação científica (o formato deve especificar quatro Es). **Resultado:** 1.234E+03
- **U:** Retorna o símbolo monetário dual "Euro" (ou outro) na posição especificada. **Exemplo:** U9999 **Resultado:** $1234
- **V:** Multiplica por 10 n vezes (n = número de 9s após V). **Exemplo:** 9999V99 **Resultado:** 123400
- **S:** Retorna o valor negativo ou positivo. **Exemplo:** S9999 **Resultado:** -1234 or +1234
- **B:** Exibe valores zero em branco, e não 0. **Exemplo:** B9999.99 **Resultado:** 1234.00

Há uma confusão muito comum entre o uso do `TO_DATE` e do `TO_NUMBER` no que diz respeito ao resultado mostrado pelo `SELECT` quando são utilizadas essas duas funções. Embora estejamos informando um formato, o Oracle não apresenta o resultado do SQL baseado nele. Isso acontece porque o formato no uso dessas funções é apenas para a conversão, e não para a visualização. Para visualizarmos o resultado com base no formato que queremos, utilizamos o `TO_CHAR`.

TO_CHAR

Função utilizada para converter tipos de dados numéricos e datas para caracteres. Além da conversão, ele é muito utilizado para formatação visual de dados.

Seguem exemplos. Aqui, são selecionados todos os nomes dos empregados e seus respectivos salários. Foi utilizada a função `TO_CHAR` para formatar os valores do salário, com o elemento de formatação para casas decimais e milhar. Note que foi utilizado o parâmetro `NLS_NUMERIC_CHARACTERS`, para definir quais caracteres devem ser utilizados para cada separador do elemento:

```
SQL> select ename
            ,to_char(sal,'9G999G999D00'
            ,'nls_numeric_characters=''.,''') sal
     from emp;

ENAME      SAL
---------- -------------
SMITH            800.00
ALLEN          1,600.00
WARD           1,250.00
JONES          3,272.50
MARTIN         1,250.00
BLAKE          2,850.00
CLARK          2,450.00
SCOTT          3,000.00
KING           5,000.00
TURNER         1,500.00
ADAMS          1,100.00
JAMES            950.00
FORD           3,000.00
MILLER         1,300.00
JOHN           1,000.00

15 rows selected.
SQL>
```

No próximo exemplo, a função `TO_CHAR` foi utilizada para formatar a data de admissão do empregado, mostrando apenas o mês referente a esta data. O `SELECT` agrupa e apresenta quantos empregados foram admitidos em cada mês:

```
SQL> select count(*) QT_ADMITIDOS
            ,to_char(hiredate,'mm') MES
     from emp
     group  by to_char(hiredate,'mm');

QT_ADMITIDOS ME
------------ --
           1 04
```

```
          2 09
          4 12
          1 11
          2 01
          2 02
          1 05
          1 03
          1 06

9 rows selected.

SQL>
```

Veja outro exemplo com `TO_CHAR`, usado na formatação de datas. Neste programa, são impressos uma data e o número do dia que ela representa no ano.

```
SQL>select '22 de agosto de 2009 será o dia '||
to_char(to_date('22/08/2009','dd/mm/yyyy'),'ddd')||
     ' do ano' from dual;

'22DEAGOSTODE2009SERÁODIA'||TO_CHAR(TO_DAT
------------------------------------------
22 de agosto de 2009 será o dia 234 do ano

SQL>
```

A seguir, são mostrados dados referentes à admissão do empregado: nome do empregado, data de admissão e dia da semana. Este último foi formatado através da função `TO_CHAR`:

```
SQL> select ename, hiredate, to_char(hiredate,'day') dia_semana
     from emp;

ENAME      HIREDATE   DIA_SEMAN
---------- ---------- ---------
SMITH      12/17/1980 wednesday
ALLEN      02/20/1981 friday
WARD       02/22/1981 sunday
```

```
JONES      04/02/1981 thursday
MARTIN     09/28/1981 monday
BLAKE      05/01/1981 friday
CLARK      06/09/1981 tuesday
SCOTT      12/09/1982 thursday
KING       11/17/1981 tuesday
TURNER     09/08/1981 tuesday
ADAMS      01/12/1983 wednesday
JAMES      12/03/1981 thursday
FORD       12/03/1981 thursday
MILLER     01/23/1982 saturday
JOHN       03/30/2011 wednesday

15 rows selected.

SQL>
```

Já neste exemplo, usamos TO_CHAR para imprimir em tela a data por extenso:

```
SQL> select 'Joinville, '||to_char(sysdate,'dd')||' de '||
initcap(to_char(sysdate, 'fmmonth'))||' de
 '||to_char(sysdate,'yyyy')||'.' from dual;

'JOINVILLE,'||TO_CHAR(SYSDATE,'DD')
-----------------------------------
Joinville, 04 de April de 2011.

SQL>
```

Aqui utilizamos a função TO_CHAR para restringir dados em um SELECT:

```
SQL> select *
     from emp
     where to_char(hiredate,'yyyy') = '1982';

    EMPNO ENAME      JOB              MGR HIREDATE          SAL
--------- ---------- --------- --------- ---------- ---------
     7788 SCOTT      ANALYST         7566 12/09/1982       3000
```

```
     7934 MILLER     CLERK           7782 01/23/1982      1300

     COMM    DEPTNO
--------- ---------
                 20
                 10

SQL>
```

Podemos concatenar strings juntamente com comandos de conversões. Veja neste exemplo, na linha 6, a conversão dos salários de todos os empregados sendo concatenada com a string `R$` representando a moeda.

```
SQL> select 'R$ '||to_char(sal,'fm9G999G990D00')
     "SAL FORMATADO", sal from emp;

SAL FORMATADO          SAL
---------------- ---------
R$ 800.00              800
R$ 1,600.00           1600
R$ 1,250.00           1250
R$ 3,272.50         3272.5
R$ 1,250.00           1250
R$ 2,850.00           2850
R$ 2,450.00           2450
R$ 3,000.00           3000
R$ 5,000.00           5000
R$ 1,500.00           1500
R$ 1,100.00           1100
R$ 950.00              950
R$ 3,000.00           3000
R$ 1,300.00           1300
R$ 1,000.00           1000

15 rows selected.

SQL>
```

O próximo exemplo se parece com o anterior. Contudo, aqui estamos utilizando os elementos de formatação `FM` e `L`. O primeiro retira os espaços

em branco, onde algum elemento de formatação não tenha sido preenchido, por ausência de valores. O segundo mostra a moeda configurada na sessão do usuário no qual o programa está sendo executado.

```
SQL> select to_char(sal,'fmL9G999G990D00') "SAL FORMATADO",
     sal from emp;

SAL FORMATADO                  SAL
----------------------- ---------
$800.00                        800
$1,600.00                     1600
$1,250.00                     1250
$3,272.50                   3272.5
$1,250.00                     1250
$2,850.00                     2850
$2,450.00                     2450
$3,000.00                     3000
$5,000.00                     5000
$1,500.00                     1500
$1,100.00                     1100
$950.00                        950
$3,000.00                     3000
$1,300.00                     1300
$1,000.00                     1000

15 rows selected.

SQL>
```

Já a seguir, utilizamos o elemento de formatação `S`, que indica o sinal referente ao valor formatado. Note que no valor positivo foi impresso, atrás do número, o sinal de positivo (+), enquanto para o valor negativo é impresso o sinal de negativo (-).

```
SQL> select to_char(174984283.75,'999,999,999.00S') positivo
           ,to_char(100-1000,'999,999,999.00S') negativo
     from dual;

POSITIVO        NEGATIVO
```

```
--------------- ---------------
174,984,283.75+         900.00-

SQL>
```

10.5 Funções condicionais

O Oracle disponibiliza outros tipos de funções, dentre as quais estão as funções condicionais. Elas são utilizadas tanto na seleção de dados pela cláusula `SELECT` como também no uso de cláusulas `WHERE`. Seu uso é bastante difundido e bem flexível. Veja a seguir:

- **DECODE:** esta estrutura funciona como uma estrutura `IF-ELSE` dentro de uma cláusula `SELECT`. É muito utilizada principalmente para visualização de dados onde é preciso realizar algum teste para saber se estes dados podem ou não aparecer;

- **NULLIF:** A função `NULLIF` compara duas expressões. Se elas forem iguais, a função retornará um valor nulo. Se elas não forem iguais, a função retornará a primeira expressão. Entretanto, você não pode especificar o literal `NULL` para a primeira expressão;

- **NVL:** para esta função são passados dois parâmetros. Se o primeiro for nulo, ele retorna o segundo, caso contrário, retorna o primeiro;

- **NVL2:** A função NVL2 examina a primeira expressão. Se a primeira expressão não for nula, a função NVL2 retornará a segunda expressão. Se a primeira expressão for nula, a terceira expressão será retornada;

- **CASE:** muito parecido com o `DECODE`. Seu objetivo também é permitir a utilização de uma estrutura tipo `IF-ELSE` dentro do comando SQL. Mas ao contrário do `DECODE` sua aplicação e visualização são mais inteligíveis (padrão *ANSI*).

- **GREATEST:** retorna a maior expressão de uma lista de valores passada como parâmetro. Todas as expressões após a primeira são convertidas para o tipo de dado da primeira antes da comparação ser realizada;

- **LEAST:** funciona ao inverso da `GREATEST`: traz a menor expressão.
- **COALESCE:** A vantagem desta função em relação ao `NVL`, é que ela pode aceitar vários valores alternativos. Por exemplo, se a primeira expressão passada para a `COALESCE` não for nula, a função retornará aquela expressão. Caso contrário, ela fará um `COALESCE` das expressões remanescentes, ou seja, vai testando as demais até encontrar uma que a satisfaça a condição de não nula.

Veja alguns exemplos. No primmeiro, utilizamos a função `CASE` para somar todos os salário por departamento, agrupando por cargo:

```
SQL> SELECT JOB,
         sum(case
              when deptno = 10 then sal
             else
               0
             end) "DEPART 10",
         sum(case
               when deptno = 20 then sal
             else
               0
             end) "DEPART 20",
         sum(case
               when deptno = 30 then sal
             else
               0
             end) "DEPART 30",
         sum(sal) "TOTAL JOB"
    FROM EMP
    GROUP BY JOB;

JOB       DEPART 10 DEPART 20 DEPART 30 TOTAL JOB
--------- --------- --------- --------- ---------
CLERK          1300      2900       950      5150
SALESMAN          0         0      4100      4100
PRESIDENT      5000         0         0      5000
VENDEDOR          0         0      1500      1500
MANAGER        2450    3272.5      2850    8572.5
```

```
ANALYST              0       6000          0       6000

6 rows selected.

SQL>
```

O segundo é idêntico ao anterior. Apenas trocamos a função `CASE` pela função `DECODE`. Vale lembrar que `CASE` é do padrão SQL *ANSI* e `DECODE` é do padrão Oracle:

```
SQL> select job
           ,sum(decode(deptno, 10, sal, 0)) "DEPART 10"
           ,sum(decode(deptno, 20, sal, 0)) "DEPART 20"
           ,sum(decode(deptno, 30, sal, 0)) "DEPART 30"
           ,sum(sal) "TOTAL JOB"
     from emp
     group  by job;

JOB       DEPART 10 DEPART 20 DEPART 30 TOTAL JOB
--------- --------- --------- --------- ---------
CLERK          1300      2900       950      5150
SALESMAN          0         0      4100      4100
PRESIDENT      5000         0         0      5000
VENDEDOR          0         0      1500      1500
MANAGER        2450    3272.5      2850    8572.5
ANALYST           0      6000         0      6000

6 rows selected.

SQL>
```

No próximo exemplo, estamos utilizando a função `NVL`. Note que quando realizamos cálculos com valores nulos (segundo comando `SELECT`) o Oracle não executa tal operação. O Oracle sempre considera como falso quando existem valores `NULL` (nulo) em cálculos aritméticos ou no uso de restrição de dados, por exemplo, em cláusulas `WHERE`. Quando isso ocorre, ele ignora a ação, mas não gera erros.

```
SQL> select * from emp;
```

```
    EMPNO ENAME      JOB              MGR HIREDATE          SAL
--------- ---------- --------- --------- ---------- ---------
     7369 SMITH      CLERK           7902 12/17/1980        800
     7499 ALLEN      SALESMAN        7698 02/20/1981       1600
     7521 WARD       SALESMAN        7698 02/22/1981       1250
     7566 JONES      MANAGER         7839 04/02/1981     3272.5
     7654 MARTIN     SALESMAN        7698 09/28/1981       1250
     7698 BLAKE      MANAGER         7839 05/01/1981       2850
     7782 CLARK      MANAGER         7839 06/09/1981       2450
     7788 SCOTT      ANALYST         7566 12/09/1982       3000
     7839 KING       PRESIDENT            11/17/1981       5000
     7844 TURNER     VENDEDOR        7698 09/08/1981       1500
     7876 ADAMS      CLERK           7788 01/12/1983       1100
     7900 JAMES      CLERK           7698 12/03/1981        950
     7902 FORD       ANALYST         7566 12/03/1981       3000
     7934 MILLER     CLERK           7782 01/23/1982       1300
     8000 JOHN       CLERK           7902 03/30/2011       1000

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10
      200        20

15 rows selected.

SQL> select sum(sal+comm) from emp;
```

```
SUM(SAL+COMM)
-------------
         9000

SQL> select sum(sal+nvl(comm,0)) from emp;

SUM(SAL+NVL(COMM,0))
--------------------
             32722.5

SQL>
```

Nestes exemplos, temos o uso das funções GREATEST e LEAST:

```
SQL> select greatest('b','x','t','u','a') maior_letra from dual;

M
-
x

SQL>

SQL> select least('b','x','t','u','a') menor_letra from dual;

M
-
a

SQL>
```

Nestes exemplos, temos o uso da função NULLIF:

```
SQL> select decode(nullif('abacaxi','abacaxi'),null,'são iguais'
     ,'são diferentes')
     comparação from dual;

COMPARAÇÃO
--------------
são iguais
```

```
SQL>

SQL> select decode(nullif('abacaxi','morango'),null,'são iguais'
     ,'são diferentes')
     comparação from dual;

COMPARAÇÃO
--------------
são diferentes

SQL>

SQL> select '10=10: ' || nvl2(nullif(10,10),'Diferentes',
                               'Iguais') resultado
  from   dual
  union all
  select '11=10: ' || nvl2(nullif(11,10),'Diferentes','Iguais')
  from   dual
  /

RESULTADO
-----------------
10=10: Iguais
11=10: Diferentes

SQL>
```

Veja o Uso da função NVL2.

```
SQL> select employee_id
        ,last_name
        ,job_id
        ,commission_pct
        ,nvl2(commission_pct,'Possui Comissão','Não Possui')
  from   employees
  where  employee_id between 175 and 183
  /

EMPLOYEE_ID LAST_NAME              JOB_ID     COMMISSION_PCT
```

```
----------- ---------------------- ---------- --------------
        175 Hutton                 SA_REP               0,25
        176 Taylor                 SA_REP               0,20
        177 Livingston             SA_REP               0,20
        178 Grant                  SA_REP               0,15
        179 Johnson                SA_REP               0,10
        180 Taylor                 SH_CLERK
        181 Fleaur                 SH_CLERK
        182 Sullivan               SH_CLERK
        183 Geoni                  SH_CLERK

NVL2(COMMISSION_PCT,'POSSUICOM
------------------------------
Possui Comissão
Possui Comissão
Possui Comissão
Possui Comissão
Possui Comissão
Não Possui
Não Possui
Não Possui
Não Possui

9 rows selected

SQL>
```

Veja um exemplo da função COALESCE onde usamos a função para recuperar as informações dos empregados, seus gerentes e suas comissões, reunindo estas informações em uma mesma coluna, dependendo da condição..

```
SQL> select last_name
        ,employee_id
        ,coalesce(to_char(commission_pct)
        ,to_char(manager_id)
        ,'No commision and no manager')
  from employees
  /
```

```
LAST_NAME                 EMPLOYEE_ID
------------------------- -----------
King                              100
Kochhar                           101
De Haan                           102
Hunold                            103
...
Partners                          146
Errazuriz                         147
Cambrault                         148
Zlotkey                           149
...

COALESCE(TO_CHAR(COMMISSION_PCT),TO_CHAR
----------------------------------------
No commision and no manager
100
100
102

.3
.3
.3
.2

107 rows selected.

SQL>
```

DECODE X CASE

Dos comandos condicionais vistos anteriormente, o `DECODE` é o mais utilizado. Ele funciona como uma espécie de condição "SE" (`IF`) para a linguagem SQL. Este comando é exclusivo do Oracle, entretanto, no padrão *ANSI* da linguagem SQL existe um comando similar chamado `CASE`. Você pode usar qualquer um deles. Entretanto, o uso do `CASE` só é permitido nas versões mais novas do banco de dados Oracle. A Oracle, nas versões mais recentes, vem inserindo comandos padrões *ANSI*, contudo, mantém seus comandos

específicos com suas características para questões de compatibilidade. Veja comparações entre esses dois comandos.

Exemplo 1

Padrão *ANSI* (também suportado pelo Oracle nas versões mais novas do banco de dados):

```
SQL> select job,
      sum(case
            when deptno = 10 then sal
          else
            0
          end) "DEPART 10",
      sum(case
            when deptno = 20 then sal
          else
            0
          end) "DEPART 20",
      sum(case
            when deptno = 30 then sal
          else
            0
          end) "DEPART 30",
      sum(sal) "TOTAL JOB"
     from emp
     group by job;

JOB        DEPART 10  DEPART 20  DEPART 30  TOTAL JOB
---------- ---------- ---------- ---------- ----------
CLERK           1300       1900        950       4150
SALESMAN           0          0       5600       5600
PRESIDENT       5000          0          0       5000
MANAGER         2450       2975       2850       8275
ANALYST            0       6000          0       6000

SQL>
```

Padrão Oracle:

```
SQL> select job,
           sum(decode(deptno, 10, sal, 0)) "DEPART 10",
```

```
         sum(decode(deptno, 20, sal, 0)) "DEPART 20",
         sum(decode(deptno, 30, sal, 0)) "DEPART 30",
         sum(sal) "TOTAL JOB"
     from  emp
     group by job;

JOB        DEPART 10  DEPART 20  DEPART 30  TOTAL JOB
--------- ---------- ---------- ---------- ----------
CLERK           1300       1900        950       4150
SALESMAN           0          0       5600       5600
PRESIDENT       5000          0          0       5000
MANAGER         2450       2975       2850       8275
ANALYST            0       6000          0       6000

SQL>
```

Exemplo 2

Padrão *ANSI* (também suportado pelo Oracle nas versões mais novas do banco de dados):

```
SQL> select ename
          , job
          , mgr
          ,
           case
             when mgr = 7902 then 'MENSALISTA'
             when mgr = 7839 then 'COMISSIONADO'
             when mgr = 7566 then 'MENSAL/HORISTA'
           else
             'OUTROS'
           end tipo
     from emp;

ENAME      JOB              MGR TIPO
---------- --------- ---------- --------------
SMITH      CLERK           7902 MENSALISTA
ALLEN      SALESMAN        7698 OUTROS
WARD       SALESMAN        7698 OUTROS
JONES      MANAGER         7839 COMISSIONADO
```

```
MARTIN     SALESMAN        7698 OUTROS
BLAKE      MANAGER         7839 COMISSIONADO
CLARK      MANAGER         7839 COMISSIONADO
SCOTT      ANALYST         7566 MENSAL/HORISTA
KING       PRESIDENT            OUTROS
TURNER     SALESMAN        7698 OUTROS
ADAMS      CLERK           7788 OUTROS
JAMES      CLERK           7698 OUTROS
FORD       ANALYST         7566 MENSAL/HORISTA
MILLER     CLERK           7782 OUTROS

14 linhas selecionadas.

SQL>
```

Padrão Oracle:

```
SQL> select ename
          ,job
          ,mgr
          ,decode(mgr,7902,'MENSALISTA'
                     ,7839,'COMISSIONADO'
                     ,7566,'MENSAL/HORISTA'
                     ,'OUTROS') tipo
     from emp;

ENAME      JOB              MGR TIPO
---------- --------- ---------- --------------
SMITH      CLERK           7902 MENSALISTA
ALLEN      SALESMAN        7698 OUTROS
WARD       SALESMAN        7698 OUTROS
JONES      MANAGER         7839 COMISSIONADO
MARTIN     SALESMAN        7698 OUTROS
BLAKE      MANAGER         7839 COMISSIONADO
CLARK      MANAGER         7839 COMISSIONADO
SCOTT      ANALYST         7566 MENSAL/HORISTA
KING       PRESIDENT            OUTROS
TURNER     SALESMAN        7698 OUTROS
ADAMS      CLERK           7788 OUTROS
JAMES      CLERK           7698 OUTROS
```

```
FORD       ANALYST        7566 MENSAL/HORISTA
MILLER     CLERK          7782 OUTROS

14 linhas selecionadas.

SQL>
```

A opção entre usar um ou outro vai depender da abrangência do seus comandos SQL. Se eles forem específicos para o uso em Oracle, o `DECODE` pode ser usado sem problemas. Inclusive, como pôde ser visto nos exemplos, ele pode se tornar visualmente mais claro para o entendimento do código. Já se suas aplicações forem abrangentes no que diz respeito a operar em vários bancos de dados, você terá que usar o padrão *ANSI*, ou seja, usar o `CASE` para que eles funcionem em qualquer banco de dados, ou pelo menos naqueles que seguem este padrão.

Capítulo 11

Integridade de dados e integridade referencial

11.1 Integridade de dados e integridade referencial

Para que as informações em um banco de dados possam permanecer íntegras, utilizamos um recurso muito interessante que o próprio banco de dados Oracle disponibiliza – estamos falando das `CONSTRAINTS`. As `CONSTRAINTS` já foram vistas no decorrer do livro, mas não foram detalhadas tecnicamente. Neste tópico iremos conhecê-las e identificá-las, tendo em vista que até este ponto já utilizamos direta ou indiretamente seus recursos.

Através das `CONSTRAINTS`, conseguimos manter, por exemplo, a integridade entre duas tabelas. Também conseguimos criticar determinado valor

que esteja sendo inserido em um campo da tabela ou até obrigar que este campo seja preenchido. Desta forma, conseguimos manter a integridade, tanto de dados como a referencial, mesmo que isso não esteja sendo feito através de um programa (telas de cadastro, por exemplo), ou seja, podemos definir regras que vão ser implementadas diretamente no banco, independente de existirem ou não controles nas aplicações.

As `CONSTRAINTS` que iremos ver são as seguintes:

- NOT NULL
- UNIQUE KEY
- PRIMARY KEY
- FOREIGN KEY
- CHECK

NOT NULL

Esta `CONSTRAINT` garante que um campo da tabela tenha que ser obrigatoriamente preenchido no momento da inclusão de algum registro.

No exemplo a seguir, a criação de uma `CONSTRAINT` já no momento da criação da tabela.

```
SQL> create table regions
     ( region_id    number      constraint region_id_nn not null
      ,region_name  varchar2(25)
     );

Table created.

SQL>
```

Veja mais exemplos a seguir para a manipulação de `CONSTRAINTS NOT NULL`:

Vamos criar uma `CONSTRAINT` de `NOT NULL` para o campo `REGION_NAME` da tabela `REGIONS`. Podemos criá-la de duas formas,

nomeando-a ou não. Utilizamos para manipular CONSTRAINTS em tabelas já existentes o comando ALTER TABLE, como vimos anteriormente. O script a seguir mostra a criação de uma CONSTRAINT nomeada, que recebe um nome no momento de sua criação.

```
SQL> desc regions
 Name                              Null?    Type
 -------------------------------- -------- ----
 REGION_ID                         NOT NULL NUMBER
 REGION_NAME                                VARCHAR2(25)

SQL> alter table regions modify region_name constraint
     region_name_nn not null;

Table altered.

SQL> desc regions;
 Name                              Null?    Type
 -------------------------------- -------- ----
 REGION_ID                         NOT NULL NUMBER
 REGION_NAME                       NOT NULL VARCHAR2(25)

SQL>
SQL> select constraint_name
     from user_cons_columns
     where table_name = 'REGIONS'
     and column_name = 'REGION_NAME';

CONSTRAINT_NAME
------------------------------
REGION_NAME_NN

SQL>
```

Note que, para visualizar a CONSTRAINT criada, montamos um SELECT em cima da tabela USER_CONS_COLUMNS, selecionando pelo nome da tabela e pelo nome da coluna.

> Assim como para as CONSTRAINTS, também podemos visualizar informações de índices no nível de coluna, através de uma tabela chamada USER_IND_COLUMNS.

Para excluir uma CONSTRAINT existente também utilizamos o comando ALTER TABLE.

```
SQL> alter table regions drop constraint region_name_nn;

Table altered.

SQL> desc regions;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 REGION_ID                                 NOT NULL NUMBER
 REGION_NAME                                        VARCHAR2(25)
SQL>
```

Já o exemplo a seguir mostra a criação de uma CONSTRAINT não nomeada. Neste caso, não quer dizer que ela não terá uma identificação, o que acontece é que o Oracle define um nome para ela. Note que a criação desta CONSTRAINT é implícita pelo Oracle, pois, em nenhum momento estamos adicionando uma CONSTRAINT. O que fazemos é modificar uma coluna NULL para NOT NULL. Fazendo isto o Oracle criará uma CONSTRAINT automaticamente.

```
SQL> alter table regions modify region_name not null;

Table altered.
SQL> desc regions;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----
 REGION_ID                                 NOT NULL NUMBER
 REGION_NAME                               NOT NULL VARCHAR2(25)

SQL> select constraint_name
     from user_cons_columns
```

```
     where table_name = 'REGIONS'
     and column_name = 'REGION_NAME';

CONSTRAINT_NAME
------------------------------
SYS_C0011019
SQL>
```

O mesmo acontece quando modificamos uma coluna para `NULL`. Neste caso o Oracle elimina a `CONSTRAINT` criada por ele.

```
SQL> alter table regions modify region_name null;

Table altered.
SQL> desc regions;
 Name                            Null?    Type
 ------------------------------- -------- ----
 REGION_ID                       NOT NULL NUMBER
 REGION_NAME                              VARCHAR2(25)

SQL> select constraint_name
     from user_cons_columns
     where table_name = 'REGIONS'
     and column_name = 'REGION_NAME';
no rows selected
SQL>
```

UNIQUE KEY

Esta `CONSTRAINT` é utilizada quando não queremos que os valores de uma coluna possam se repetir. Seu conceito é bem parecido com a chave primária. Entretanto, veremos mais adiante que a chave primária, além de não permitir valores repetidos, não aceita valores nulos nos campos que fazem parte da chave. No caso da `CONSTRAINT UNIQUE`, valores nulos são permitidos, ou seja, ela desconsidera estes valores em sua verificação. Veja o exemplo:

```
SQL> desc regions;
 Name                            Null?    Type
```

```
 -------------------------------- -------- ----
 REGION_ID                        NOT NULL NUMBER
 REGION_NAME                               VARCHAR2(25)

SQL> alter table regions modify region_name constraint
     region_name_un unique;

Table altered.

SQL> desc regions;
 Name                             Null?    Type
 -------------------------------- -------- ----
 REGION_ID                        NOT NULL NUMBER
 REGION_NAME                               VARCHAR2(25)

SQL> select constraint_name
     from user_cons_columns
     where table_name = 'REGIONS'
     and column_name = 'REGION_NAME';

CONSTRAINT_NAME
------------------------------
REGION_NAME_UN

SQL>
```

Agora vamos tentar inserir um registro, colocando no campo REGION_NAME algum nome já existente na tabela:

```
SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa

SQL> insert into regions values (5,'Europe');
```

```
insert into regions values (5,'Europe')
*
ERROR at line 1:
ORA-00001: unique constraint (TSQL2.REGION_NAME_UN) violated


SQL>
```

Como o nome `Europe` já existe na tabela, é gerado um erro. Isso ocorreu devido ao fato de termos criado uma `CONSTRAINT` do tipo `UNIQUE`. O mesmo não acontece quando tentamos inserir um valor nulo para a coluna em questão:

```
SQL> insert into regions values (5,null);

1 row created.

SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa
        5

SQL>
```

Para desabilitar ou excluir uma `CONSTRAINT`, utilizamos o comando `ALTER TABLE`.

```
SQL> alter table regions disable constraint region_name_un;

Table altered.

SQL> alter table regions drop constraint region_name_un;

Table altered.
```

```
SQL>
```

Uma `CONSTRAINT` também pode ser criada através do seguinte comando:

```
SQL> alter table regions add constraint region_name_un_2
     unique (region_name);

Table altered.

SQL>
```

PRIMARY KEY

Como já foi explicado anteriormente, a chave primária possui um objetivo bem definido. Ela garante que a informação contida na coluna que faz parte desta chave seja única. Dentro de uma tabela, ela vai servir como um identificador único para cada linha existente. Uma chave primária pode ser composta, ou seja, pode ser constituída de várias colunas. Nesse caso, essas colunas servirão de identificadores únicos para cada registro.

Exemplo de chave primária simples:

```
SQL> alter table regions add constraint reg_id_pk primary key
     (region_id);

Table altered.

SQL>
```

Exemplo de chave primária composta:

```
SQL> alter table regions add constraint reg_pk_01 primary key
     (region_id, region_name);
alter table regions add constraint reg_pk_01 primary key
(region_id, region_name)
                                                 *
ERROR at line 1:
ORA-02260: table can have only one primary key
```

```
SQL>
```

Note que ao tentarmos criar a `CONSTRAINT` de chave primária o Oracle gerou um erro nos dizendo que já existe uma chave primária criada para a tabela `REGIONS`. É isso mesmo, o Oracle está certo. Criamos uma chave primária chamada `REG_ID_PK`, no exemplo anterior. Uma tabela só pode conter uma chave primária. Desta forma, se quisermos mesmo criar uma nova chave primária, devemos primeiramente excluir a existente.

```
SQL> alter table regions drop constraint reg_id_pk

Table altered.

SQL>
```

Vamos tentar novamente:

```
SQL> alter table regions add constraint reg_pk_01 primary key
     (region_id, region_name);
alter table regions add constraint reg_pk_01 primary key
(region_id, region_name)
                                                               *
ERROR at line 1:
ORA-01449: column contains NULL values; cannot alter to NOT NULL


SQL>
```

Ops! Outro erro ocorreu. O Oracle diz que não podemos criar uma chave primária utilizando colunas que possuem valores nulos. Isso é verdade e faz parte das premissas para a criação de uma chave primária. Lembra quando criamos uma região sem nome nos exemplos anteriores? Agora temos que excluir este registro ou atualizá-lo dando um nome de região.

Vamos excluí-lo:

```
SQL> select * from regions
     /
```

```
REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa
        5

SQL> delete from regions where region_id = 5;

1 row deleted.

SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa

SQL>
```

Agora sim!

```
SQL> alter table regions add constraint reg_pk_01 primary key
     (region_id, region_name);

Table altered.

SQL>
```

Vamos testar se nossa chave funciona:

```
SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Europe
```

```
         2 Americas
         3 Asia
         4 Middle East and Africa

SQL> insert into regions (region_id, region_name) values
     (1,'Brazil/West');

1 row created.

SQL>

SQL> insert into regions (region_id, region_name)
     values (1,'Europe');
insert into regions (region_id, region_name) values (1,'Europe')
*
ERROR at line 1:
ORA-00001: unique constraint (TSQL2.REGION_NAME_UN_2) violated


SQL> insert into regions (region_id, region_name)
     values (1,null);
insert into regions (region_id, region_name) values (1,null)
                                                       *
ERROR at line 1:
ORA-01400: cannot insert NULL into
("TSQL2"."REGIONS"."REGION_NAME")


SQL> select * from regions;

REGION_ID REGION_NAME
--------- -------------------------
        1 Brazil/West
        1 Europe
        2 Americas
        3 Asia
        4 Middle East and Africa

SQL> insert into regions (region_id, region_name)
```

```
     values (5,'Brazil/South');

1 row created.

SQL>
```

Vejamos algumas características:

No Oracle, quando criamos uma chave primária, automaticamente, um índice único é criado. Caso já exista um índice, para a coluna ou colunas que compõem esta chave, o Oracle utilizará o existente. Entretanto, para que isto aconteça, todas as colunas deste índice devem estar presentes na composição da chave primária, inclusive na mesma ordem em que são apresentadas na chave.

Quando excluímos uma chave primária, o índice a ela associado também é excluído. Se quisermos excluir um índice referente a uma chave primária, devemos primeiramente desabilitá-la. Já quando ativamos uma chave primária, um índice único é criado, caso ele não exista.

Vamos analisar a tabela `DEPARTMENTS`:

```
select index_name name
      ,'INDEX '||index_type type
      ,null referencia
      ,status
from   user_indexes
where  table_name = 'DEPARTMENTS'
union
select constraint_name
      ,'CONSTRAINT '||upper(decode(constraint_type,'C', 'Check',
        'O', 'R/O View',
        'P', 'Primary',
        'R', 'Foreign',
        'U', 'Unique',
        'V','Check view')) constraint_type
      ,'CONSTRAINT '||r_constraint_name
      ,status
from   user_constraints
where  table_name = 'DEPARTMENTS'
order  by 1
```

Este SELECT traz informações tanto de CONSTRAINTS como de índices. Veja o resultado na sequência:

NAME	TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	ENABLED
DEPT_ID_PK	INDEX NORMAL		VALID
DEPT_LOCATION_IX	INDEX NORMAL		VALID
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.1

Nas linhas em destaque temos para a tabela DEPARTMENTS um índice e uma CONSTRAINT de PRIMARY, ou seja, uma chave primária foi criada para esta tabela. Automaticamente, quando ela foi criada, o Oracle também criou um índice para esta chave. Vamos ver o que acontece quando tentamos excluir ou desabilitar a chave:

```
SQL> alter table departments disable constraint DEPT_ID_PK;
alter table departments disable constraint DEPT_ID_PK
*
ERROR at line 1:
ORA-02297: cannot disable constraint (TSQL.DEPT_ID_PK) -
dependencies exist


SQL>
```

Primeiramente, o Oracle nos informa que não podemos desabilitar a CONSTRAINT de chave primária, pois existem pendências provindas dela. O que o Oracle quer dizer com isto? Provavelmente, esta tabela possui alguns dados e eles já estão sendo utilizados em uma ou mais tabelas. Mas como o Oracle identificou isto? Não se preocupe, não tem mistério. Se formos procurar quais são as tabelas que usam como referencia a nossa chave primária, vamos encontrar o resultado a seguir.

Nas linhas em destaque, vemos que existem duas chaves estrangeiras (veremos o conceito no próximo tópico) que referenciam, ou seja, ligam a tabela DEPARTAMENTS às tabelas EMPLOYEES e JOB_HISTORY.

```
SQL> select table_name
           ,constraint_name
     from user_constraints
     where r_constraint_name = 'DEPT_ID_PK';

TABLE_NAME                     CONSTRAINT_NAME
------------------------------ ------------------------------
EMPLOYEES                      EMP_DEPT_FK
JOB_HISTORY                    JHIST_DEPT_FK

SQL>
```

Com os comandos que aprendemos anteriormente, vamos desabilitá-las:

```
SQL> alter table employees disable constraint EMP_DEPT_FK;

Table altered.

SQL> alter table job_history disable constraint JHIST_DEPT_FK;

Table altered.

SQL>
```

Em seguida tentamos novamente, desabilitar a CONSTRAINT de chave primária:

```
SQL> alter table departments disable constraint DEPT_ID_PK;

Table altered.

SQL>
```

Pronto! CONSTRAINTS desabilitadas. Mas talvez você deve estar se perguntando: Não há jeito mais fácil de desabilitar estas referências, sem ser uma a uma? Há sim! Você pode fazer isso colocando logo após o nome da

CONSTRAINT a seguinte cláusula: CASCADE. Ao invés de desabilitarmos as CONSTRAINTS uma a uma, vamos desabilitar todas as CONSTRAINTS (de uma só vez) que fazem referência à chave primária em questão, utilizando CASCADE. Veja o exemplo:

```
SQL> alter table departments disable constraint DEPT_ID_PK
     cascade;

Table altered.

SQL>
```

Isso faz com que o Oracle procure as dependências existentes para a CONSTRAINT em questão (chave primária) e desabilite qualquer referência ligada a ela.

Agora vamos executar novamente o SELECT e ver o resultado:

NAME	TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	DISABLED
DEPT_ID_PK	INDEX NORMAL		VALID
DEPT_LOCATION_IX	INDEX NORMAL		VALID
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.2

O índice ainda permanece e está válido. Entretanto, nossa CONSTRAINT de chave primária foi desabilitada. Vamos a mais um caso, voltamos a habilitar a nossa chave primária. O CASCADE não funciona para habilitarmos as CONSTRAINTS em cascata, neste caso vamos habilitar uma a uma. Mas espere, vamos continuar trabalhando com a chave primária, então, vejamos os comandos na sequência. Vamos executar somente o primeiro, os demais ficam somente como exemplo.

```
SQL>
SQL> alter table departments enable constraint DEPT_ID_PK
```

```
     /

Table altered.

SQL> alter table job_history enable constraint JHIST_DEPT_FK
     /

Table altered.

SQL> alter table employees enable constraint EMP_DEPT_FK
     /

Table altered.

SQL>
```

Voltamos a ter o status anterior às alterações.

NAME	TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	ENABLED
DEPT_ID_PK	INDEX NORMAL		VALID
DEPT_LOCATION_IX	INDEX NORMAL		VALID
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.3

Agora vamos excluir o índice `DEPT_ID_PK` para ver como vai se comportar nossa chave primária. Vale lembrar que, para excluir um índice referente a uma chave primária, temos que desabilitá-la primeiro.

```
SQL> alter table departments disable constraint DEPT_ID_PK;

Table altered.

SQL> drop index DEPT_ID_PK;
```

```
Index dropped.

SQL>
```

Segue o novo resultado:

NAME	TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	DISABLED
DEPT_LOCATION_IX	INDEX NORMAL		VALID
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.4

Podemos ver que o índice foi excluído e que a `CONSTRAINT` de chave primária continua existindo, embora desabilitada. Vamos habilitá-la para ver o que acontece:

```
SQL> alter table departments enable constraint DEPT_ID_PK;

Table altered.

SQL>
```

Resultado:

NAME	TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	ENABLED
DEPT_ID_PK	INDEX NORMAL		VALID
DEPT_LOCATION_IX	INDEX NORMAL		VALID
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.5

Note que, quando habilitamos a CONSTRAINT de chave primária, o Oracle criou o índice. Preste atenção em um detalhe. Como já excluímos o índice uma vez, o Oracle vai excluir o índice sempre que desabilitarmos a CONSTRAINT.

```
SQL> alter table departments disable constraint DEPT_ID_PK;

Table altered.

SQL>
```

Resultado:

NAME	TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	DISABLED
DEPT_LOCATION_IX	INDEX NORMAL		VALID
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.6

Com estes exemplos, ficou mais fácil entender como funciona o conceito de como o Oracle mantém a integridade das chaves primárias. Agora vamos entender como funcionam as chaves estrangeiras.

FOREIGN KEY

Seu papel é estabelecer relacionamentos entre tabelas e um banco de dados. Quando escrevermos um modelo de dados envolvendo tabelas, nós pensamos em relacionamentos, que são responsáveis por ligar uma ou mais tabelas. Um relacionamento pode ocorrer entre duas ou mais tabelas, mas também pode acontecer dentro da mesma tabela. Chamamos isto de autorrelacionamento.

Mas um relacionamento não serve apenas para melhor visualizarmos o modelo de dados ou para mostrar como determinadas tabelas estão se relacionando. Seu objetivo é muito mais importante e merece total atenção.

Através dos relacionamentos, podemos manter algo que é chamado de **integridade referencial**, ou seja, garantias de que os dados entre as tabelas que estão se relacionando estejam íntegros. Isso é conseguido através da criação das `FOREIGN KEYS` ou Chaves Estrangeiras, como são comumente chamadas.

Existem regras para definição e funcionamento dessas chaves, por isto, quando estivermos definindo um modelo de dados relacional, devemos prestar bastante atenção no negócio como um todo. Precisamos que de ponta a ponta os dados possam permanecer íntegros, de forma que possamos recuperá-los da maneira como quisermos e quando nós precisarmos.

Depois de bem definidas e criadas na base de dados, seu funcionamento segue regras para que o banco de dados possa identificar e avisar caso algum problema aconteça. Basicamente, seu funcionamento ocorre da seguinte forma: a coluna `chave estrangeira` sempre referencia uma coluna `chave primária`, sendo de outra tabela ou da mesma. Quando referenciamos, não apenas indicamos uma ligação, mas também criamos uma dependência entre os objetos, ou melhor, entre seus registros. Dessa forma, para manter esse controle funcionando, as seguintes regras e restrições são aplicadas:

- Não é permitida a inclusão de dados em uma coluna chave estrangeira que não existam em uma chave primária, tanto no que diz respeito a relacionamentos entre duas ou mais tabelas quanto em um autorrelacionamento.

- Assim como na inserção, em uma atualização de dados em uma chave estrangeira, também não são permitidos valores que não constam na chave primária referenciada.

- No caso de exclusão de dados de uma chave primária, a integridade referencial estabelecida pela chave estrangeira garante que os registros referenciados não possam ser excluídos, evitando que os registros fiquem órfãos.

- O mesmo acontece quando tentamos atualizar dados de uma chave primária que possuem referências com outras chaves estrangeiras. A lei da

integridade referencial garante que isso não seja possível.

Vamos verificar as CONSTRAINTS criadas para a tabela DEPARTMENTS:

```
select constraint_name
      ,'CONSTRAINT '||upper(decode(constraint_type,
                                    'C', 'Check',
                                  'O', 'R/O View',
                                  'P', 'Primary',
                                  'R', 'Foreign',
                                  'U', 'Unique',
                                  'V','Check view'))
                                  constraint_type
      ,'CONSTRAINT '||r_constraint_name referencia
      ,status
from   user_constraints
where  table_name = 'DEPARTMENTS'
order  by 1
```

CONSTRAINT_NAME	CONSTRAINT_TYPE	REFERENCIA	STATUS
DEPT_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	ENABLED
DEPT_LOC_FK	CONSTRAINT FOREIGN	CONSTRAINT LOC_ID_PK	ENABLED
DEPT_MGR_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
DEPT_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.7

Observe que a tabela DEPARTMENTS possui várias CONSTRAINTS. Vamos destacar a CONSTRAINT DEPT_ID_PK que é a CONSTRAINT de chave primária da tabela.

Vamos verificar as CONSTRAINTS criadas para a tabela EMPLOYEES.

```
select constraint_name
      ,'CONSTRAINT '||upper(decode(constraint_type,
                                    'C', 'Check',
                                  'O', 'R/O View',
                                  'P', 'Primary',
```

```
                                    'R', 'Foreign',
                                    'U', 'Unique',
                                    'V','Check view'))
                                    constraint_type
        ,'CONSTRAINT '||r_constraint_name referencia
        ,status
from    user_constraints
where   table_name = 'EMPLOYEES'
order   by 1
```

CONSTRAINT_NAME	CONSTRAINT_TYPE	REFERENCIA	STATUS
EMP_DEPT_FK	CONSTRAINT FOREIGN	CONSTRAINT DEPT_ID_PK	ENABLED
EMP_EMAIL_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED
EMP_EMAIL_UK	CONSTRAINT UNIQUE	CONSTRAINT	ENABLED
EMP_EMP_ID_PK	CONSTRAINT PRIMARY	CONSTRAINT	ENABLED
EMP_HIRE_DATE_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED
EMP_JOB_FK	CONSTRAINT FOREIGN	CONSTRAINT JOB_ID_PK	ENABLED
EMP_JOB_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED
EMP_LAST_NAME_NN	CONSTRAINT CHECK	CONSTRAINT	ENABLED
EMP_MANAGER_FK	CONSTRAINT FOREIGN	CONSTRAINT EMP_EMP_ID_PK	ENABLED
EMP_SALARY_MIN	CONSTRAINT CHECK	CONSTRAINT	ENABLED

Fig. 11.8

A tabela EMPLOYEES também possui várias CONSTRAINTS, até mais que a tabela anterior. Isso ocorre porque em nosso modelo de dados esta tabela possui muitos relacionamentos com outras do sistema. Vamos nos atentar para duas CONSTRAINTS, a EMP_EMP_ID_PK e a EMP_DEPT_FK. A primeira é referente à chave primária da tabela e a segunda faz referência à tabela DEPARTMENTS, ou seja, nosso relacionamento. Se analisarmos o contexto, essa chave estrangeira garante que só conseguimos cadastrar um empregado para um departamento existente na tabela de departamentos. Com isso, não corremos o risco de cadastrar empregados com códigos de departamentos inexistentes. Está aí nossa integridade referencial. Vamos ver se as regras e restrições mencionadas anteriormente são verdadeiras?

Vamos tentar inserir um novo empregado para um departamento que não

existe na tabela de departamentos.

```
SQL> insert into employees
     values(207,'FERNANDO','SILVA','FSILVA','9998-9990',SYSDATE,
     'MK_MAN',1000,0,206,
            280,'01-JAN-79',SYSDATE);
insert into employees
*
ERROR at line 1:
ORA-02291: integrity constraint (TSQL.EMP_DEPT_FK) violated -
parent key not found


SQL>
```

Através da CONSTRAINT de chave estrangeira EMP_DEPT_FK, foi garantida a integridade dos dados. O Oracle identificou a violação da integridade referencial e bloqueou a inserção. Vamos tentar atualizar o departamento de um determinado empregado utilizando um departamento inexistente, para ver como o Oracle vai se comportar agora:

```
SQL> update employees
     set department_id = 390
     where employee_id = 100;
update employees
*
ERROR at line 1:
ORA-02291: integrity constraint (TSQL.EMP_DEPT_FK) violated -
parent key not found


SQL>
```

Mais uma vez, através da chave estrangeira a integridade foi mantida. Agora vamos fazer o contrário. Já vimos que através da chave estrangeira criada para a tabela referenciada (EMPLOYEES) garantimos a integridade dos dados, não permitindo, com a CONSTRAINT EMP_DEPT_FK, que fossem utilizados códigos de departamentos não existentes. Contudo, precisamos testar as duas pontas e, para isto, vamos tentar realizar algumas alterações

na tabela referência (DEPARTMENTS), onde está localizada a chave primária DEPT_ID_PK.

Vimos que os valores contidos em colunas de chave estrangeira devem ser originários de uma determinada chave primária pertencente a outra tabela ou até mesmo à própria tabela. Em nossos exemplos, estamos trabalhando com o relacionamento entre as tabelas EMPLOYEES e DEPARTMENTS que estão ligadas pela coluna DEPARTMENT_ID. Vamos tentar alterar algum código de departamento existente que contenha empregados cadastrados para ver o que acontece.

```
SQL> update departments
     set department_id = 999
     where department_id = 10
     /
update departments
*
ERROR at line 1:
ORA-02292: integrity constraint (TSQL.EMP_DEPT_FK) violated -
child record found


SQL>
```

Note que a consistência foi feita e a mensagem de que o registro que está sendo alterado possui dependência em outras fontes foi mostrada. Mais uma vez, foi garantida a integridade. Na exclusão ocorrerá a mesma situação, contudo, vamos ver o seguinte exemplo:

```
SQL> delete from departments
     where department_id = 10;
delete from departments
*
ERROR at line 1:
ORA-02292: integrity constraint (TSQL.EMP_DEPT_FK) violated -
child record found


SQL>
```

Agora vejamos um exemplo de criação de CONSTRAINT de FOREIGN KEY:

```
SQL> alter table emp add constraint dept_emp_fk
     foreign key (deptno)
     references dept (deptno);

Tabela alterada.

SQL>
```

Neste exemplo, é vista a criação da FOREIGN KEY em cima da tabela EMP, a qual possui o campo DEPTNO que faz referência à tabela DEPT, campo, este, utilizado na chave. Note também que, ao criar esse tipo de chave, somos obrigados a informar esta referência, utilizando a cláusula REFERENCES como forma de consistir o relacionamento entre as duas tabelas.

CHECK

Como o nome sugere, as CONSTRAINTS de CHECK são utilizadas para a realização de checagens em colunas de uma tabela. Você as utiliza para determinar valores iniciais e finais para campos data e numérico, para restringir a entrada de determinados valores etc. Por exemplo, vamos supor que para o campo valor da comissão não é permitida a entrada de valores caso o empregado não seja um vendedor. Através de uma CONSTRAINT CHECK pode-se fazer este controle. Outro exemplo seria termos um campo que só pode receber os valores 'M' e 'F'.

Veja alguns exemplos:

Criando

Permitir código de departamento entre 10 e 900:

```
SQL> alter table departments add constraint dept_id_ck
        check (department_id between 10 and 900);

Table altered.

SQL>
```

Permitir somente a entrada de valores em maiúsculo para o campo EMAIL:

```
SQL> alter table em loyees add constraint email_ck
        check (email = upper(email));

Table altered.

SQL>
```

Permitir somente a inserção de determinadas regiões:

```
SQL> alter table regions add constraint region_name_ck
        check (region_name in ('Europe','Americas','Asia',
                                 'Middle East and Africa'));

Table altered.

SQL>
```

Permitir somente a entrada de registros apenas quando a soma do salário mais a comissão seja maior que dez mil.

```
SQL> alter table emp add constraint sal_comm_ck
        check (sal + nvl(comm,0) < 10000);

Table altered.

SQL>
```

Não permitir que empregados não vendedores (SALESMAN) tenham comissão informadas.

```
SQL> alter table emp add constraint sal_comm_sales_ck
         check ( (nvl(comm,0) = 0 and job <> 'SALESMAN')
          or (job = 'SALESMAN'));

Table altered.

SQL>
```

Excluindo

```
SQL> alter table emp drop constraint sal_comm_ck;

Table altered.

SQL>
```

Desabilitando / Habilitando

```
SQL> alter table emp disable constraint sal_comm_sales_ck;

Table altered.

SQL> alter table emp enable constraint sal_comm_sales_ck;

Table altered.

SQL>
```

A criação e manipulação das `CONSTRAINTS CHECK` são muito parecidas com as já vistas normalmente. Seu funcionamento também é muito intuitivo e de fácil entendimento. Vamos ver agora algumas aplicações, em cima dos exemplos vistos.

Entrada incorreta de código de departamento:

```
SQL> insert into departments values(901,'TESTE CHECK',200,1700);
insert into departments values(901,'TESTE CHECK',200,1700)
*
ERROR at line 1:
ORA-02290: check constraint (TSQL.DEPT_ID_CK) violated


SQL>
```

Entrada incorreta do e-mail:

```
SQL> insert into employees
       values( 900,'Pato','Donald','pdonald','234.423.4567',
              '03-JAN-2001','IT_PROG',9000,0,102,60,
```

```
                '03-JAN-1975','03-NOV-1991');
insert into employees
*
ERROR at line 1:
ORA-02290: check constraint (TSQL.EMAIL_CK) violated


SQL>
```

Entrada incorreta da região:

```
SQL> insert into regions values(5,'TESTE');
insert into regions values(5,'TESTE')
*
ERROR at line 1:
ORA-02290: check constraint (TSQL.REGION_NAME_CK) violated


SQL>
```

Entrada incorreta do salário mais comissão (valor maior):

```
SQL> insert into emp values(9875,'MARCOS','SALESMAN',7698,
     '09-JUL-1989',7500,4000,20);
insert into emp values(9875,'MARCOS','SALESMAN',7698,
'09-JUL-1989',7500,4000,20)
*
ERROR at line 1:
ORA-02290: check constraint (TSQL.SAL_COMM_CK) violated

SQL>
```

Entrada incorreta da comissão quanto ao cargo:

```
SQL> insert into emp values(9875,'MARCOS','CLERK',7902,
     '09-JUN-1985',8000,3000,20);
insert into emp values(9875,'MARCOS','CLERK',7902,
'09-JUN-1985',8000,3000,20)
*
ERROR at line 1:
```

```
ORA-02290: check constraint (TSQL.SAL_COMM_SALES_CK) violated

SQL>
```

Podemos ver que todas as CONSTRAINTS criadas obtiveram sucesso nos seus objetivos, garantindo a integridade das informações. Não restam dúvidas de que sua utilidade pode ajudar e muito na concretização e bom funcionamento do modelo de dados.

FOREIGN KEY DELETE CASCADE, SET NULL E DISABLE

O Oracle dispõe de algumas funcionalidades que podem ser usadas na criação de CONSTRAINTS. Vamos discuti-las na sequência, mas, antes disso, vamos criar uma cópia da tabela DEPT. Vamos chamá-la de DEPARTAMENTO. Ela será utilizada em nossos exemplos.

```
SQL> create table departamento as select * from dept;

Table created.

SQL> select * from departamento;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
50 TI             BRASIL

SQL>
SQL> alter table departamento add constraint pk_deptno
     primarykey (deptno);

Table altered.
SQL>
```

DELETE CASCADE

Quando definimos uma CONSTRAINT de FOREIGN KEY como DELETE CASCADE, estamos informando ao Oracle que, quando um valor de chave primária referente à chave estrangeira em questão for excluído, os registros referenciados por esta chave (em outras tabelas ou na mesma) podem ser excluídos. Embora este recurso seja muito interessante e prático, deve-se tomar muito cuidado, pois precisamos estar ciente de todas as referências existentes para não acabarmos por excluir registros que não poderiam ser excluídos.

```
SQL> create table empregado
       ( empno        number(4,0)
        ,ename        varchar2(10)
        ,job          varchar2(9)
        ,mgr          number(4,0)
        ,hiredate     date
        ,sal          number(7,2)
        ,comm         number(7,2)
        ,deptno       number(2,0) constraint deptno_fk references
  departamento(deptno) on delete cascade
       )
     /

Table created.

SQL>
```

A CONSTRAINT DEPTNO_FK diz ao Oracle que, se algum departamento da tabela DEPARTAMENTO for excluído, todos os empregados que estão locados neste departamento devem ser excluídos também. Vamos ver como isso funcionaria:

Para começar, inserimos na tabela EMPREGADO os dados da tabela EMP:

```
SQL> insert into empregado select * from emp;

14 rows created.

SQL>
```

Mostrando os dados da tabela populada:

```
SQL> select * from empregado;

    EMPNO ENAME      JOB              MGR HIREDATE        SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80       800
     7499 ALLEN      SALESMAN        7698 20-FEB-81      1600
     7521 WARD       SALESMAN        7698 22-FEB-81      1250
     7566 JONES      MANAGER         7839 02-APR-81    3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81      1250
     7698 BLAKE      MANAGER         7839 01-MAY-81      2850
     7782 CLARK      MANAGER         7839 09-JUN-81      2450
     7788 SCOTT      ANALYST         7566 09-DEC-82      3000
     9999 KING       PRESIDENT            17-NOV-81      5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81      1500
     7876 ADAMS      CLERK           7788 12-JAN-83      1100
     7900 JAMES      CLERK           7698 03-DEC-81       950
     7902 FORD       ANALYST         7566 03-DEC-81      3000
     7934 MILLER     CLERK           7782 23-JAN-82      1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10

14 rows selected.

SQL>
```

Mostrando os dados da tabela DEPARTAMENTO:

```
SQL> select * from departamento;

   DEPTNO DNAME          LOC
--------- -------------- -------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON
       50 TI             BRASIL

SQL>
```

Agora vamos tentar excluir o departamento 20:

```
SQL> delete from departamento where deptno = 20;

1 row deleted.

SQL>
```

Lembre-se que, quando criamos a tabela EMPREGADOS, definimos uma CONSTRAINT DELETE CASCADE. Logo, quando excluímos um registro na tabela DEPARTAMENTO, todas as referências com a tabela EMPREGADOS são afetados. Veja o resultado a seguir:

```
SQL> select * from empregado;

    EMPNO ENAME      JOB             MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7499 ALLEN      SALESMAN       7698 20-FEB-81       1600
     7521 WARD       SALESMAN       7698 22-FEB-81       1250
     7654 MARTIN     SALESMAN       7698 28-SEP-81       1250
     7698 BLAKE      MANAGER        7839 01-MAY-81       2850
     7782 CLARK      MANAGER        7839 09-JUN-81       2450
     9999 KING       PRESIDENT           17-NOV-81       5000
     7844 TURNER     VENDEDOR       7698 08-SEP-81       1500
     7900 JAMES      CLERK          7698 03-DEC-81        950
     7934 MILLER     CLERK          7782 23-JAN-82       1300
```

```
     COMM     DEPTNO
--------- ---------
      300         30
      500         30
     1400         30
                  30
                  10
                  10
        0         30
                  30
                  10

9 rows selected.

SQL>
```

Como pode ser visto, além da exclusão do departamento, também foram excluídos os empregados deste departamento na tabela `EMPREGADOS`.

SET NULL

Já quando definimos esta `CONSTRAINT` como `SET NULL`, estamos indicando que, para as referências existente nos valores de chave estrangeira, deve-se atualizar com valores nulos, caso os valores de chaves primárias forem excluídos. Ao contrário do `DELETE CASCADE`, os registros referenciados não serão excluídos, apenas terão seus valores atualizados para nulos.

```
SQL> create table empregado
       ( empno        number(4,0) constraint empno_pk primary key
        ,ename        varchar2(10)
        ,job          varchar2(9)
        ,mgr          number(4,0)  constraint mgr_fk references
           empregado on delete set null
        ,hiredate     date
        ,sal          number(7,2)
        ,comm         number(7,2)
        ,deptno       number(2,0)
       );
```

```
Table created.

SQL>
```

Neste exemplo estamos definindo que caso um gerente seja excluído, os empregados que estão subordinados a ele ficarão sem gerente, ou seja, ficarão com o campo `MGR` como nulo. Assim, são preservados os registros e a integridade dos dados é mantida. Veja a seguir um exemplo:

```
SQL> insert into empregado select * from emp;

14 rows created.

SQL>
```

Visualizando os dados da tabela `EMPREGADOS`:

```
SQL> select * from empregado;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
```

```
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10

14 rows selected.

SQL>
```

Vamos excluir um empregado, que tenha pelo menos um subordinado. Conseguimos visualizar isso através da coluna `MGR`:

```
SQL> delete from empregado where empno = 7839;

1 row deleted.

SQL> select * from empregado;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- ---------- --------- ----------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER              02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER              01-MAY-81       2850
     7782 CLARK      MANAGER              09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
```

```
     7900 JAMES      CLERK          7698 03-DEC-81        950
     7902 FORD       ANALYST        7566 03-DEC-81       3000
     7934 MILLER     CLERK          7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
        0        30
                 20
                 30
                 20
                 10

13 rows selected.

SQL>
```

Excluímos o empregado KING. Logo, seu subordinado JONES teve a informação referente à gerência limpa.

DISABLE

Utilizando a expressão DISABLE, estamos indicando que a CONSTRAINT deve ser criada, mas que deve permanecer desabilitada. O contrário não é necessário, pois quando criamos uma CONSTRAINT ela por padrão já é ativada.

```
SQL> create table empregado
       ( empno        number(4,0)
        ,ename        varchar2(10)
        ,job          varchar2(9)
        ,mgr          number(4,0)
        ,hiredate     date
        ,sal          number(7,2) constraint ck_sal
       check (sal< 10000) disable
```

```
        ,comm         number(7,2)
        ,deptno       number(2,0)
       );

Table created.

SQL>
```

Criamos uma `CONSTRAINT` que controla a entrada dos valores na coluna salário. Contudo, a deixamos desabilitada.

Vamos verificar seu funcionamento.

Alimentando a tabela `EMPREGADO`:

```
SQL> insert into empregado select * from emp;

14 rows created.

SQL> commit;

Commit complete.

SQL>
```

Visualizando os dados inseridos:

```
SQL> select * from empregado;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
```

```
     7876 ADAMS      CLERK          7788 12-JAN-83       1100
     7900 JAMES      CLERK          7698 03-DEC-81        950
     7902 FORD       ANALYST        7566 03-DEC-81       3000
     7934 MILLER     CLERK          7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10

14 rows selected.

SQL>
```

Atualizando o salário referente ao empregado `7839` para `20000`. Nenhum erro ocorreu, ou seja, a `CONSTRAINT` não foi acionada.

```
SQL> update empregado set sal = 20000 where empno = 7839;

1 row updated.

SQL> rollback;

Rollback complete.

SQL>
```

Agora vamos habilitar a `CONSTRAINT` para ver o que acontece.

```
SQL> alter table empregado enable constraint ck_sal;

Table altered.

SQL>
```

Executamos o mesmo `UPDATE` e agora a `CONSTRAINT` foi acionada, gerando o erro:

```
SQL> update empregado set sal = 20000 where empno = 7839;
update empregado set sal = 20000 where empno = 7839
*
ERROR at line 1:
ORA-02290: check constraint (TSQL.CK_SAL) violated


SQL>
```

Veja agora como poderiam ficar todas as declarações em uma definição de tabela:

```
SQL> create table empregado
       ( empno      number(4,0)  constraint pk_empno primary key
        ,ename      varchar2(10)
        ,job        varchar2(9)
        ,mgr        number(4,0)  constraint fk_mgr references
  empregado on delete set null
        ,hiredate   date         default sysdate
        ,sal        number(7,2)  constraint ck_sal
  check (sal< 10000) disable
        ,comm       number(7,2)  default 0
        ,deptno     number(2,0)  constraint fk_deptno references
   departamento (deptno) on delete cascade
       );

Table created.

SQL>
```

Vamos abrir um parâmetro para falar sobre a declaração de criação das CONSTRAINTS. Nos exemplos de criação de CONSTRAINTS sempre vimos como criá-las através do comando ALTER TABLE. Todavia, as CONSTRAINTS também podem ser criadas e definidas no momento em que estamos montando o SCRIPT da criação da tabela. Também há duas formas de fazer isto. A Oracle chama essas definições de CONSTRAINT IN-LINE e CONSTRAINT OUT-OF-LINE.

CONSTRAINT INLINE

Este tipo consiste em declarar as CONSTRAINTS logo depois do nome da coluna. Você pode fazer isso de forma nomeada ou não. Como já foi visto anteriormente, caso não nomeie, o Oracle criará um nome interno para a CONSTRAINT. Veja os dois exemplos:

Nomeando-as:

```
create table employees
( employee_id     number(6,0)  constraint pk_emp primary key
 ,first_name      varchar2(20)
 ,last_name       varchar2(25) constraint nn_last_name check
 ("LAST_NAME" IS NOT NULL)
 ,email           varchar2(25) constraint nn_email NOT NULL
 ,phone_number    varchar2(20)
 ,hire_date          date         constraint nn_hire_date check
 ("HIRE_DATE" IS NOT NULL)
 ,job_id          varchar2(10) constraint nn_emp_job check
 ("JOB_ID" IS NOT NULL)
 ,salary          number(8,2)
 ,commission_pct  number(2,2)
 ,manager_id      number(6,0)
 ,department_id   number(4,0)  constraint fk_emp_dept references
 departments (department_id)
  )
```

Sem nomeação:

```
create table employees
( employee_id       number(6,0)  primary key
```

```
 ,first_name          varchar2(20)
 ,last_name           varchar2(25) check
 ("LAST_NAME" IS NOT NULL)
 ,email               varchar2(25) NOT NULL
 ,phone_number        varchar2(20)
 ,hire_date           date         check ("HIRE_DATE" IS NOT NULL)
 ,job_id              varchar2(10) check ("JOB_ID" IS NOT NULL)
 ,salary              number(8,2)
 ,commission_pct      number(2,2)
 ,manager_id          number(6,0)
 ,department_id       number(4,0)  references departments
 (department_id)
)
```

A questão de nomear ou não vai de cada um. As duas formas estão corretas. Contudo, se você quiser usar padrões de nomes e tornar mais fácil e ágil a identificação das CONSTRAINTS criadas no seu banco de dados, fica aí a dica. Apenas vale ressaltar que, independente de dar nomes ou não, isso não prejudica em nada seu funcionamento.

CONSTRAINT OUFOFLINE

Já este tipo consiste em declarar as CONSTRAINTS logo depois da declaração de todas as colunas. Neste caso, a forma nomeada é obrigatória. Veja o exemplo:

```
create table employees
( employee_id            number(6,0)
 ,first_name             varchar2(20)
 ,last_name              varchar2(25)
 ,email                  varchar2(25) constraint nn_email not null
 ,phone_number           varchar2(20)
 ,hire_date              date
 ,job_id                 varchar2(10)
 ,salary                 number(8,2)
 ,commission_pct         number(2,2)
 ,manager_id             number(6,0)
 ,department_id          number(4,0)
 ,constraint pk_emp primary key (employee_id)
```

```
 ,constraint nn_last_name check ("LAST_NAME" IS NOT NULL)
 ,constraint nn_hire_date check ("HIRE_DATE" IS NOT NULL)
 ,constraint nn_emp_job check ("JOB_ID" IS NOT NULL)
 ,constraint fk_emp_dept foreign key (department_id) references
 departments (department_id)
)
```

Para as CONSTRAINTS de NOT NULL não é possível declarar neste formato, mantendo, portanto, o formato INLINE.

Capítulo 12

Oracle avançado

12.1 Trabalhando com views

Muitas vezes os usuários precisam acessar várias colunas que estão em mais de uma tabela. Isso pode se tornar complexo quando não há o domínio da linguagem SQL e dispendioso quando se precisa desta informação várias vezes. Com o comando `SELECT` podemos criar um conjunto de dados que satisfaça as necessidades do usuário, montando grupos de informações provenientes de uma ou mais fontes (tabelas). Essa prática denomina-se `VIEW`. Utilizando `VIEWS`, pode-se criar uma montagem dos dados, com somente aquelas informações de que o usuário precisa. Portanto, uma `VIEW` nada mais é que uma montagem de dados, criada em tempo de execução, como se fosse o resultado de um `SELECT`. Veja lgumas observações sobre as `VIEWS`:

- Geralmente, as visões não são armazenadas, precisando ser recalculadas para cada consulta que as referencie. No banco é guardada apenas sua especificação.
- A declaração `SELECT` que define uma visão não pode conter uma cláusula `ORDER BY`.
- Podem-se fazer `INSERTS` em `VIEWS`, desde que em sua especificação não haja `SUBQUERIES`, `UNIONS`, `MINUS`, mais de uma tabela na cláusula `FROM`, não haja `SELECTS` em cláusulas `FROM` e `WHERE`, e não tenham funções de agregação ou expressões.
- As `VIEWS` também podem ser do tipo "materializada". Neste caso, os dados são armazenados no banco de dados.
- Este tipo de `VIEW` comumente é utilizado quando grandes volumes de dados são recuperados através dela.
- É possível executar operações `DML` normalmente nas `VIEWS` simples.
- Não é possível remover linhas se a `VIEW` contiver: funções de grupo, cláusula `GROUP BY`, `DISTINCT` ou `ROWNUM`.
- Não é possível modificar os dados em uma `VIEW` se ela contiver: funções de grupo, cláusula `GROUP BY`, `DISTINCT`, `ROWNUM` ou colunas definidas por expressões.
- Não será possível adicionar dados usando uma `VIEW` se ela incluir: funções de grupo, cláusula `GROUP BY`, `DISTINCT`, `ROWNUM`, colunas definidas por expressões ou colunas `NOT NULL` em tabelas base que não sejam selecionadas pela `VIEW`.

Criando uma view de empregados por departamento que recebem comissão

`SELECT` base para a construção da `VIEW`:

```
SQL> select dname, ename, job, sal, comm
     from emp e, dept d
     where e.deptno = d.deptno
```

```
     and e.comm is not null;

DNAME          ENAME      JOB              SAL       COMM
-------------- ---------- --------- ---------- ----------
SALES          ALLEN      SALESMAN        1600        300
SALES          WARD       SALESMAN        1250        500
SALES          MARTIN     SALESMAN        1250       1400
SALES          TURNER     VENDEDOR        1500          0
RESEARCH       JOHN       CLERK           1000        200

SQL>
```

Criação da VIEW EMP_DEPT_COMM:

```
SQL> create view emp_dept_comm as
     select dname, ename, job, sal, comm
     from emp e, dept d
     where e.deptno = d.deptno
     and e.comm is not null;

View created.
SQL>
```

DESCRIPTION e SELECT da VIEW.

```
SQL> desc emp_dept_comm;
 Name                             Null?    Type
 -------------------------------- -------- ----
 DNAME                                     VARCHAR2(14)
 ENAME                                     VARCHAR2(10)
 JOB                                       VARCHAR2(9)
 SAL                                       NUMBER(7,2)
 COMM                                      NUMBER(7,2)

SQL> select * from emp_dept_comm;

DNAME          ENAME      JOB              SAL       COMM
-------------- ---------- --------- ---------- ----------
SALES          ALLEN      SALESMAN        1600        300
SALES          WARD       SALESMAN        1250        500
```

```
SALES          MARTIN     SALESMAN       1250       1400
SALES          TURNER     VENDEDOR       1500          0
RESEARCH       JOHN       CLERK          1000        200

SQL>
```

Criando uma view de empregados que trabalham com Tecnologia da Informação (IT) nos Estados Unidos

SELECT base para a construção da VIEW:

```
SQL> select e.first_name, e.job_id
     from   employees e
           ,departments d
           ,locations l
     where e.department_id = d.department_id
     and d.location_id = l.location_id
     and l.country_id = 'US'
     and d.department_name = 'IT';

FIRST_NAME           JOB_ID
-------------------- ----------
Alexander            IT_PROG
Bruce                IT_PROG
David                IT_PROG
Valli                IT_PROG
Diana                IT_PROG

SQL>
```

Criação da VIEW EMP_IT_US:

```
SQL> create view emp_it_us as
     select e.first_name, e.job_id
     from   employees e
           ,departments d
           ,locations l
     where e.department_id = d.department_id
     and d.location_id = l.location_id
     and l.country_id = 'US'
```

```
     and d.department_name = 'IT';

View created.

SQL>
```

DESCRIPTION e SELECT da VIEW:

```
SQL> desc emp_it_us
 Name                               Null?    Type
 ------------------------------- -------- ----
 FIRST_NAME                                  VARCHAR2(20)
 JOB_ID                             NOT NULL VARCHAR2(10)

SQL> select * from emp_it_us;

FIRST_NAME           JOB_ID
-------------------- ----------
Alexander            IT_PROG
Bruce                IT_PROG
David                IT_PROG
Valli                IT_PROG
Diana                IT_PROG

SQL>
```

Criando VIEWS complexas para mostrar dados resultantes de um comando SQL contendo funções de grupo, juntando duas ou mais tabelas:

```
SQL> create or replace view dept_sum_vu (name, minsal, maxsal,
     avgsal) as select d.department_name, min(e.salary),
     max(e.salary), avg(e.salary) from   employees e JOIN
     departments d on (e.department_id = d.department_id)
     group   by d.department_name
     /

View created.

SQL> select * from dept_sum_vu;
```

```
NAME                              MINSAL     MAXSAL     AVGSAL
------------------------------ ---------- ---------- ----------
Administration                    6442.04    6442.04    6442.04
Accounting                       12152.03    17569.2  14860.615
Executive                         24889.7    35138.4 28305.9333
IT                                6149.22    13176.9   8433.216
Purchasing                        3660.25    16105.1   6076.015
Human Resources                   9516.65    9516.65    9516.65
Public Relations                    14641      14641      14641
Shipping                          3074.61   12005.62 5088.56089
Finance                          10102.29    17569.2   12591.26
Sales                             8931.01    20497.4 13112.3074
Marketing                          8784.6    19033.3   13908.95

11 rows selected.

SQL>
```

Alterando uma view

Para alterar uma `VIEW`, recrie-a utilizando a expressão `OR REPLACE`.

```
SQL> create or replace view emp_it_us as
     select e.first_name, e.job_id
     from   employees e
           ,departments d
           ,locations l
     where e.department_id = d.department_id
     and d.location_id = l.location_id
     and l.country_id = 'US'
     and e.manager_id = 103 --inclusão de um gerente específico.
     and d.department_name = 'IT';

View created.

SQL>
```

Excluindo uma view

```
SQL> drop view emp_dept_comm;

View dropped.

SQL>
```

Usando a cláusula WITH CHECK OPTION

Quando estamos trabalhando com VIEWS podemos nos assegurar quanto às questões de integridade. Por exemplo, se quisermos criar restrições para determinados dados, que não possam ser manipulados através de comandos DML, podemos utilizar a cláusula WITH CHECK OPTION para garantir isso. Veja o exemplo a seguir:

```
SQL> create view emp_manager124 as
  2  select first_name, manager_id
  3  from   employees
  4  where  manager_id = 124
  5  with check option constraint emp_manager124
  6  /

View created.

SQL> update emp_manager124 set manager_id = 201 where
     manager_id = 124;
update emp_manager124 set manager_id = 201 where
manager_id = 124
       *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation


SQL>
```

Neste exemplo, criamos uma VIEW que seleciona todos os empregado cujo o gerente seja o 124. Note que na definição da VIEW utilizamos a cláusula WITH CHECK OPTION. Veja, também, que utilizamos a palavra-chave CONSTRAINT e definimos um nome para ela. Feito isto, quando tentamos atualizar dados cujo gerente não seja aquele com o código 124, por exemplo,

um erro de violação é gerado. Esse tipo de checagem também funcionará se tentarmos inserir ou excluir empregados que não tenham como valor para o campo MANAGER_ID o código 124. Dessa forma, dados que não são selecionados pela VIEW não podem sofrer qualquer tipo de alteração se utilizarmos esta cláusula.

VIEWS somente Leitura

Outra opção bastante interessante é a possibilidade de restringirmos a VIEW para que nenhum comando DML possa ter ação sobre ela, ou seja, mantê-la em modo somente leitura. Para isso, utilizamos a cláusula WITH READ ONLY. Veja o exemplo a seguir:

```
SQL> create view emp_manager124 as
  2  select first_name, manager_id
  3  from   employees
  4  where  manager_id = 124
  5  with read only;

View created.

SQL> update emp_manager124 set manager_id = 201 where
     manager_id = 124;
update emp_manager124 set manager_id = 201 where
manager_id = 124
                                                    *
ERROR at line 1:
ORA-42399: cannot perform a DML operation on a read-only view


SQL>
```

Note, neste exemplo, que ao tentarmos atualizar dados da VIEW, uma mensagem de erro é gerada informando que se trata de uma VIEW em modo somente leitura.

VIEWS Materializadas (Materialized VIEWS)

As *Views Materialized*, assim conhecidas desde a versão 9i, são nada mais que objetos do banco de dados Oracle que contêm dados locais de tabelas remotas ou são usadas pra criar uma agregação de tabelas em um determinado intervalo de tempo. Uma curiosidade é que nas versões precedentes à 9i as Views Materializadas eram conhecidas como *Snapshots*.

Vimos que as `VIEWS` comuns não guardam os dados propriamente ditos, guardam apenas a estrutura de como estes dados devem ser recuperados. Já as Views Materialized funcionam de forma diferente. Elas são geralmente utilizadas quando estamos trabalhando com grandes quantidades de dados ou quando estes dados demandam um processamento muito demorado, que, se fosse executado online, poderia comprometer o uso de todo o sistema.

A criação de uma View Materialized envolve toda uma configuração que deve ser feita na sua definição. É neste momento que você especifica, por exemplo, quando ela vai ser atualizada, de que forma vai ser atualizada e assim por diante. Não vamos explorá-la a fundo devido à necessidade de conhecimentos mais apurados, inclusive, sobre a administração do banco Oracle. A título de curiosidade, segue um exemplo:

```
SQL> create materialized view emp_dept_20
     refresh complete next sysdate + 1
     as select * from emp where deptno = 20;

Snapshot created.

SQL> select * from emp_dept_20;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- ---------- ----------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7902 FORD       ANALYST         7566 03-DEC-81       3000

     COMM    DEPTNO
```

```
--------- ---------
                 20
                 20
                 20
                 20
                 20

SQL> drop materialized view emp_dept_20;

Snapshot dropped.

SQL>
```

Esta VIEW terá seus resultados atualizados a cada 1 dia. Veja o exemplo a seguir:

```
SQL> select sysdate from dual;

SYSDATE
---------
16-APR-11

SQL>
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
```

```
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM     DEPTNO
--------- ---------
                  20
      300         30
      500         30
                  20
     1400         30
                  30
                  10
                  20
                  10
        0         30
                  20
                  30
                  20
                  10

14 rows selected.

SQL>
```

Neste quadro, mostramos a data corrente e, na sequência, todos os registros da tabela `EMP`.

```
SQL> select * from emp_dept_20;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     COMM     DEPTNO
--------- ---------
                  20
```

```
                20
                20
                20
                20

SQL> insert into empvalues(555,'EDUARDO','ANALISTA',NULL
     ,'01-JAN-2011',5000,NULL,20);

1 row created.

SQL> commit;

Commit complete.

SQL>
```

Logo após, selecionamos os dados da `VIEW`. Na sequência, fazemos um `INSERT` na tabela `EMP`.

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- ---------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
...
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300
      555 EDUARDO    ANALISTA             01-JAN-11       5000

     COMM     DEPTNO
--------- ----------
                  20
      300         30
```

```
      500          30
                   20
     1400          30

                   20
                   30
                   20
                   10
                   20

15 rows selected.

SQL> select * from emp_dept_20;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- ---------- ----------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7902 FORD       ANALYST         7566 03-DEC-81       3000

     COMM     DEPTNO
---------- ----------
                   20
                   20
                   20
                   20
                   20
SQL>
```

Selecionamos os dados da tabela `EMP` e constatamos que o registro inserido está lá. Contudo, note que, embora tenhamos inserido um empregado novo, ele não aparece na `VIEW`. Isso acontece porque nossa `VIEW` está programada para atualizar uma vez por dia apenas.

Vamos ver como ela fica um dia depois da entrada do empregado novo:

```
SQL> select sysdate from dual;
```

```
SYSDATE
---------
17-APR-11

SQL> select * from emp_dept_20;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7902 FORD       ANALYST         7566 03-DEC-81       3000
      555 EDUARDO    ANALISTA             01-JAN-11       5000

     COMM    DEPTNO
--------- ---------
                 20
                 20
                 20
                 20
                 20
                 20

6 rows selected.
SQL>
```

Após 24 horas, nossa `VIEW` foi atualizada e agora o registro está sendo mostrado.

12.2 Trabalhando com índices

Pode-se dizer que índices são estruturas de dados mantidas internamente pelo gerenciador do banco de dados para otimizar o acesso aos dados nas tabelas. Seu principal propósito é aumentar a performance na busca pelos dados em tabelas do banco de dados. Existem algumas regras para que eles sejam utilizados nessas buscas:

- Quando as colunas na cláusula SELECT fazem parte de um índice, e pelo menos uma destas colunas exista na cláusula WHERE.
- Quando as colunas na cláusula WHERE aparecem sem o emprego de funções, ou seja, não fazem parte de uma expressão, ou não são passadas por parâmetro em funções tais como TO_CHAR, TRUNC etc., a menos que sejam índices baseados em função.
- No caso de índices compostos, a falta de uma coluna na cláusula WHERE pode fazer com que o banco não use o índice, ou seja, o Oracle leva em consideração a posição das colunas obedecendo a ordem definida no momento da criação do índice.
- Caso a quantidade de dados a ser retornada em um comando SELECT seja muito grande, o Oracle pode optar por não usar o índice.

No Oracle, os índices são classificados por tipos. Abordaremos os tipos **b-tree** e **bitmap**. Os índices b-tree são os índices de uso geral no Oracle. Eles são os tipos de índices padrão criados durante a criação de índices. Os índices b-tree podem ser índices de uma única coluna (simples) ou índices compostos (várias colunas). Índices b-tree podem ter até 32 colunas. Os índices deste tipo podem ser únicos ou não únicos.

Único

Este tipo de índice é utilizado em colunas que não possuem valores em duplicidade. Caso tenhamos uma coluna com valores duplicados, esta coluna não poderá ser um índice único, a menos que seja criado um índice composto. Neste caso, as colunas juntas, que formam o índice, não poderão ter valores duplicados. Veja a tabela EMP:

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
```

```
     7566 JONES      MANAGER        7839 02-APR-81    3272.5
     7654 MARTIN     SALESMAN       7698 28-SEP-81      1250
     7698 BLAKE      MANAGER        7839 01-MAY-81      2850
     7782 CLARK      MANAGER        7839 09-JUN-81      2450
     7788 SCOTT      ANALYST        7566 09-DEC-82      3000
     7839 KING       PRESIDENT           17-NOV-81      5000
     7844 TURNER     VENDEDOR       7698 08-SEP-81      1500
     7876 ADAMS      CLERK          7788 12-JAN-83      1100
     7900 JAMES      CLERK          7698 03-DEC-81       950
     7902 FORD       ANALYST        7566 03-DEC-81      3000
     7934 MILLER     CLERK          7782 23-JAN-82      1300
     8000 JOHN       CLERK          7902 30-MAR-11      1000

     COMM    DEPTNO
--------- ---------
               20
      300      30
      500      30
               20
     1400      30
               30
               10
               20
               10
        0      30
               20
               30
               20
               10
      200      20

15 rows selected.
SQL>
```

Agora, vamos criar um índice único para a coluna `EMPNO`. Note que ela não possui valores duplicados, logo, é uma forte candidata a este tipo de índice.

```
SQL> create unique index empno_un1 on emp (empno);

Index created.

SQL>
```

Veja o que acontece quando tentamos criar um índice único para a coluna `JOB`:

```
SQL> create unique index job_un1 on emp (job);
create unique index job_un1 on emp (job)
                                *
ERROR at line 1:
ORA-01452: cannot CREATE UNIQUE INDEX; duplicate keys found


SQL>
```

Não conseguimos, pois na coluna `JOB` existem valores duplicados. Contudo, se adicionarmos mais uma coluna, sendo que os valores concatenados destas colunas nunca sejam duplicados (índices compostos), veremos que nosso índice é criado com sucesso.

```
SQL> create unique index job_ename_u1 on emp (job, ename);

Index created.

SQL>
```

Os índices também garantem a integridade dos dados e, no caso dos índices únicos, ele não deixará que valores duplicados sejam inseridos ou atualizados nos campos compostos por ele. Vamos verificar se isto é verdade?

```
SQL> insert into emp (empno) values (7369);
insert into emp (empno) values (7369)
*
ERROR at line 1:
ORA-00001: unique constraint (TSQL.EMPNO_UN1) violated
```

```
SQL>
```

Na atualização ocorre o mesmo erro.

```
SQL> update emp set empno = 7369
     where empno = 7839;
update emp set empno = 7369
*
ERROR at line 1:
ORA-00001: unique constraint (TSQL.EMPNO_UN1) violated


SQL>
```

Este controle de integridade também ocorre com os índices compostos:

```
SQL> insert into emp (empno,job, ename)
     values (999,'CLERK','SMITH');
insert into emp (empno,job, ename) values (999,'CLERK','SMITH')
*
ERROR at line 1:
ORA-00001: unique constraint (TSQL.JOB_ENAME_U1) violated


SQL>
```

Se tomarmos cuidado e inserirmos os dados corretamente, respeitando os índices, nossos comandos executarão dentro do esperado.

```
SQL> insert into emp (empno) values (8888);

1 row created.

SQL>

SQL> update emp set empno = 9999
     where empno = 7839;

1 row updated.

SQL>
```

```
SQL> insert into emp (empno, job, ename)
     values (999,'CLERK','JOAOZINHO');

1 row created.

SQL>
```

Note neste último comando que, embora o valor CLERK já exista na coluna JOB, ele não acusa erro de chave violada. Isto ocorre porque o índice não é composto apenas pelo campo JOB, mas também pelo campo ENAME. Logo, os valores CLERK e JOAOZINHO não existem na tabela EMP, de modo que não violamos o índice único.

Não único

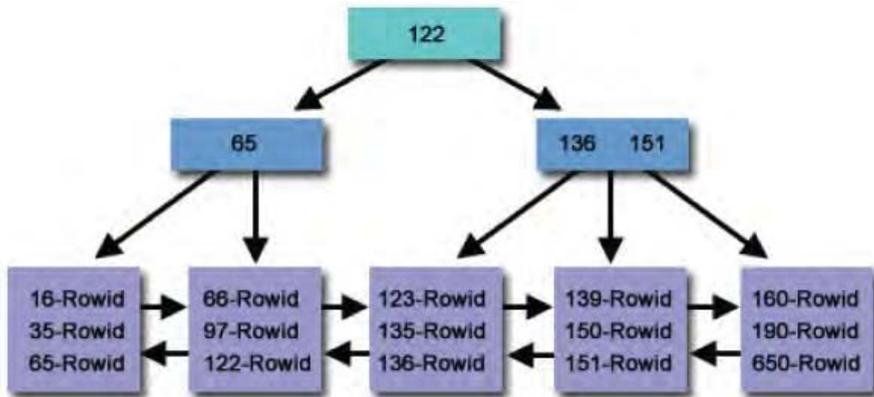
Ao contrário do índice único, este índice permite que sejam criados índices em colunas cujos valores sejam duplicados. Isso vale tanto para índice simples (com uma coluna), como para índices compostos (uma ou mais colunas).

```
SQL> create index job_un1 on emp (job);

Index created.

SQL>

SQL> create index job_deptno_in1 on emp (job, deptno);

Index created.

SQL>
```

Logo, o controle de integridade não é realizado.

```
SQL> insert into emp (empno, ename, job)
     values (1111,'PEDRINHO','SALESMAN');

1 row created.

SQL>
```

```
SQL> insert into emp (empno, ename, job, deptno)
     values (1112,'LUIZINHO','SALESMAN',30);

1 row created.

SQL>
```

Veja a estrutura de um índice `b-TREE`, definida por grupos de `ROWIDS` de forma balanceada.

BITMAP

Em um índice bitmap, o Oracle cria um bitmap para cada chave distinta e depois armazena as `ROWIDS` associadas ao valor de chave como um bitmap. No caso de um índice comum, este processo é realizado por meio do armazenamento de uma lista de `ROWIDS` para cada chave de índice que corresponde às linhas da tabela com aquele valor de chave, pois, no Oracle, cada chave é armazenada juntamente com o `ROWID` correspondente. O funcionamento do índice bitmap acontece da seguinte forma:

- Um mapa de bits para cada valor de chave é usado em vez de uma lista de `ROWIDS`. Cada bit corresponde a uma possível `ROWID` e, se o bit es-

tiver marcado isso, significa que a linha com o ROWID correspondente contém o valor chave.

- Uma função de mapeamento é usada para converter a posição do bit para o valor correspondente do ROWID, para que o índice forneça a mesma funcionalidade que um índice comum, mesmo usando uma representação diferente.

Uma das razões para a utilização dos índices bitmaps é quando temos tabelas que possuem uma cardinalidade baixa, isto é, a coluna (ou colunas) na qual o índice é baseado possui um número pequeno de valores distintos (por exemplo: sexo, estado civil, UF). No caso da tabela EMP, temos um campo que se encaixa nesses aspectos, o campo JOB. Veja a seguir:

```
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     9999 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
```

```
                    20
      1400          30
                    30
                    10
                    20
                    10
         0          30
                    20
                    30
                    20
                    10

14 rows selected.

SQL>
```

Note que na coluna `JOB` diversos valores se repetem e com isto os dados não se tornam muito distintos. Vejamos a criação de um índice bitmap para esta coluna:

```
SQL> create bitmap index job_ibm1 on emp (job);

Index created.

SQL>
```

Também podemos criar índices compostos do tipo bitmap.

Veja a estrutura de um índice bitmap, onde são definidos mapas conforme a coluna ou colunas escolhidas para a indexação.

	7369 SMITH (ROWID)	7499 ALLEN (ROWID)	7521 WARD (ROWID)	7566 JONES (ROWID)	7654 MARTIN (ROWID)	7698 BLAKE (ROWID)	7782 CLARK (ROWID)	7788 SCOTT (ROWID)	7839 KING (ROWID)	7844 TURNER (ROWID)	7876 ADAMS (ROWID)	7900 JAMES (ROWID)	7902 FORD (ROWID)	7934 MILLER (ROWID)
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
SALESMAN	0	1	1	0	1	0	0	0	0	1	0	0	0	0
PRESIDENT	0	0	0	0	0	0	0	0	1	0	0	0	0	0
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
ANALYST	0	0	0	0	0	0	0	1	0	0	0	0	1	0

Fig. 12.2

Outro exemplo:

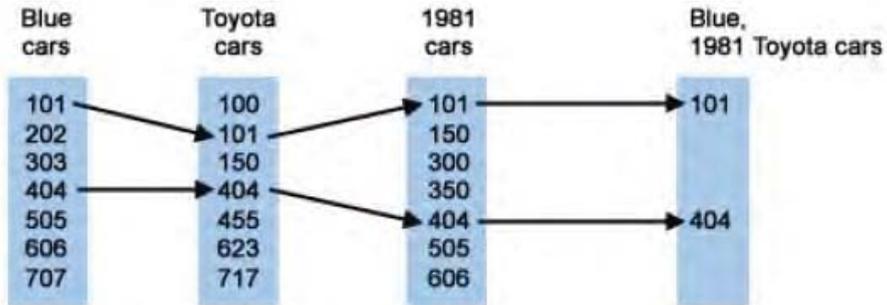



Fig. 12.3: Quando criamos uma CONSTRAINT de chave primária (visto mais adiante) o Oracle, automaticamente, cria um índice único para as colunas desta chave, caso não exista.

> Quando criamos uma `CONSTRAINT` de chave primária (veja mais adiante) o Oracle, automaticamente, cria um índice único para as colunas desta chave, caso não exista.
>
> Se você tentar criar um índice, sendo ele b-TREE ou bitmap, para uma combinação de colunas já existente, o Oracle não deixará realizar esta criação. Ele mostrará uma mensagem informando que a lista de colunas já está indexada.

Diretrizes para criação de índices

- **Crie um índice quando:**
  - Uma coluna contiver uma ampla faixa de valores;
  - Uma coluna contiver um grande número de valores nulos;
  - Uma ou mais colunas forem usadas conjuntamente em uma cláusula `WHERE` ou em uma condição de junção;
  - A tabela for grande e houver a expectativa de que a maioria das consultas recuperará menos de 2% a 4% das linhas na tabela.

- **Não crie um índice quando:**
  - As colunas não forem usadas com frequência como uma condição na consulta;
  - A tabela for pequena e houver a expectativa de que a maioria das consultas recuperará mais de 2% a 4% das linhas na tabela;
  - A tabela for atualizada com frequência;
  - As colunas indexadas forem referenciadas como parte de uma expressão.

12.3 Sinônimos

Para simplificar o acesso aos objetos em um banco de dados utilizamos sinônimos. Através deles, é possível abreviar nomes de objetos longos e referir-se facilmente a objetos de outro usuário. Um sinônimo funciona como sendo um alias para o objeto, podendo ser criado de forma privada embaixo do próprio `SCHEMA` ou em nível público. Este último dá acesso a todos os usuários, entretanto, só pode ser criado ou excluído pelo administrador do banco de dados.

Como já foi dito, o sinônimo privado deve ser criado embaixo do `SCHEMA` (usuário) o qual deseja acessar o objeto. Contudo, não podemos confundir seu objetivo. Mesmo tendo um sinônimo criado para um determinado objeto, isso não garantirá o acesso. Lembre-se de que os sinônimos são `ALIAS` (apelidos) e não fornecem concessões a objetos.

Antes de criarmos os sinônimos, precisamos ter concessões (`GRANTS`) de acesso aos objetos, para nossos sinônimos funcionarem.

Os sinônimos públicos (`PUBLIC`) são criados também embaixo de um `SCHEMA`, entretanto, ele não estará restrito a ele, como acontece com a criação privada. Ao contrário, ele estará disponível para todos os usuários que possuem acesso ao banco.

Nos capítulos anteriores, vimos como dar `GRANTS` aos objetos para que outros usuários possam acessar objetos dos quais não são donos. Vimos que, embora determinado usuário tivesse `GRANT` para acessar um objeto, era necessário colocar na frente do nome deste objeto o nome do seu dono. Através

da criação de sinônimos, podemos isentar esta informação. Com isso, o usuário que for acessar o objeto não precisa saber embaixo de qual SCHEMA ele está criado.

Vamos acompanhar alguns exemplos. Para isso, utilizaremos o usuário TSQL já visto em capítulos anteriores. Também vamos criar mais dois usuários para testarmos as funcionalidades dos sinônimos.

Criando usuários:

```
SQL> create user aluno1 identified by aluno1;

User created.

SQL> grant resource, connect to aluno1;

Grant succeeded.

SQL>

SQL> create user aluno2 identified by aluno2;

User created.

SQL> grant resource, connect to aluno2;

Grant succeeded.

SQL>
```

Testando usuários:

```
SQL> conn aluno1/aluno1@xe2;
Connected.
SQL>
SQL> conn aluno2/aluno2@xe2;
Connected.
SQL>
```

Selecionando os dados da tabela de empregados através do usuário TSQL, que é seu dono:

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10
```

```
14 rows selected.
SQL>
```

Vamos ver o que acontece quando tentamos acessar a tabela `EMP` através do usuário `ALUNO1`, que criamos recentemente.

```
SQL> conn aluno1/aluno1@xe2;
Connected.
SQL>
SQL> select * from emp;
select * from emp
              *
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> select * from tsql.emp;
select * from tsql.emp
                   *
ERROR at line 1:
ORA-00942: table or view does not exist

SQL>
```

Vimos que não conseguimos acessar a tabela, nem mesmo colocando o nome do usuário dono na frente do nome. Vamos tentar criar um sinônimo para a tabela, ainda no usuário `ALUNO1`:

```
SQL> create synonym emp for tsql.emp;
create synonym emp for tsql.emp
*
ERROR at line 1:
ORA-01031: insufficient privileges


SQL>
```

Não foi possível criar o sinônimo, pois `ALUNO1` não tem privilégios para isso. Vamos voltar ao usuário `TSQL` e criar este sinônimo por lá. Outra alternativa seria dar `GRANT` de criação de sinônimos para `ALUNO1`.

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> create synonym aluno1.emp for tsql.emp;

Synonym created.

SQL>
```

Agora vamos voltar ao usuário `ALUNO1` e tentar acessar novamente:

```
SQL> conn aluno1/aluno1@xe2;
Connected.
SQL>
SQL> select * from emp;
select * from emp
              *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
```

Ainda não estamos conseguindo acessar a tabela. Não é para tanto: apenas criamos o sinônimo, ainda não foi dada a concessão de acesso à tabela. Lembre-se do que falamos anteriormente – sinônimos não concedem acesso. Vamos criar uma concessão, através do comando `GRANT` para a tabela em questão. Para isso, temos que estar conectado no usuário dono da tabela ou um usuário que tenha acesso total ao banco de dados (`SYS`, `SYSTEM` ou um usuário que tenha recebido o `GRANT` de `DBA`, por exemplo). Mas, para brincarmos um pouco com o cenário, vamos excluir o sinônimo criado anteriormente antes de criar o `GRANT`:

```
SQL> drop synonym emp;

Synonym dropped.

SQL>
```

Perfeito. Agora vamos criar o GRANT:

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL>
SQL>
SQL> grant select on emp to aluno1;

Grant succeeded.

SQL>
```

Criada a concessão, vamos tentar acessar a tabela:

```
SQL> conn aluno1/aluno1@xe2;
Connected.
SQL>
SQL> select * from emp;
select * from emp
              *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> select * from tsql.emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- ---------- ----------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
```

```
     7900 JAMES      CLERK          7698 03-DEC-81        950
     7902 FORD       ANALYST        7566 03-DEC-81       3000
     7934 MILLER     CLERK          7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
                 20
                 30
                 20
                 10

14 rows selected.

SQL>
```

Percebemos que na primeira tentativa não foi possível acessá-la, entretanto, ao colocar no nome do dono na frente da tabela, conseguimos visualizar seus dados. Agora, para que não seja preciso informar este dono, vamos criar nosso sinônimo novamente:

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> create synonym aluno1.emp for tsql.emp;

Synonym created.

SQL> conn aluno1/aluno1@xe2;
Connected.
```

```
SQL>
SQL> select * from emp;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL
---------- ---------- ---------- ---------- --------- ----------
      7369 SMITH      CLERK           7902 17-DEC-80        800
      7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
      7521 WARD       SALESMAN        7698 22-FEB-81       1250
      7566 JONES      MANAGER         7839 02-APR-81     3272.5
      7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
      7698 BLAKE      MANAGER         7839 01-MAY-81       2850
      7782 CLARK      MANAGER         7839 09-JUN-81       2450
      7788 SCOTT      ANALYST         7566 09-DEC-82       3000
      7839 KING       PRESIDENT            17-NOV-81       5000
      7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
      7876 ADAMS      CLERK           7788 12-JAN-83       1100
      7900 JAMES      CLERK           7698 03-DEC-81        950
      7902 FORD       ANALYST         7566 03-DEC-81       3000
      7934 MILLER     CLERK           7782 23-JAN-82       1300

      COMM     DEPTNO
---------- ----------
                   20
       300         30
       500         30
                   20
      1400         30
                   30
                   10
                   20
                   10
         0         30
                   20
                   30
                   20
                   10

14 rows selected.
```

```
SQL>
```

Pronto. Sinônimo criado e testado. Que tal agora modificarmos este sinônimo, simplificando seu nome? Para não ficarmos alternando entre um usuário e outro, vamos dar `GRANT` para `ALUNO1` criar sinônimos.

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> grant create synonym to aluno1;

Grant succeeded.

SQL>
```

Agora, vamos excluir e criar novamente o sinônimo modificando-o:

```
SQL> conn aluno1/aluno1@xe2;
Connected.
SQL>
SQL> drop synonym aluno1.emp;

Synonym dropped.

SQL> create synonym aluno1.e for tsql.emp;

Synonym created.

SQL> select * from e;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
```

```
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM     DEPTNO
--------- ---------
                20
      300       30
      500       30
                20
     1400       30
                30
                10
                20
                10
        0       30
                20
                30
                20
                10

14 rows selected.

SQL>
```

Para que ALUNO2 possa acessar esta mesma tabela, também são necessários os mesmos passos realizados para o ALUNO1, a menos, é claro, que criemos um sinônimo público (PUBLIC). Entretanto, temos que ter ciência que todos os usuários criados na base de dados terão acesso a este sinônimo. Contudo, apenas os que possuem concessão terão acesso aos dados da tabela. Vamos criar um sinônimo PUBLIC:

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> create public synonym funcionarios for tsql.emp;
```

```
Synonym created.

SQL>
```

Vamos acessá-lo através de `ALUNO1` e `ALUNO2`:

```
SQL> conn aluno1/aluno1@xe2;
Connected.
SQL>
SQL> select * from funcionarios;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300


     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
```

```
                10
        0       30
                20
                30
                20
                10

14 rows selected.

SQL>

SQL> conn aluno2/aluno2@xe2;
Connected.
SQL>
SQL> select * from funcionarios;
select * from funcionarios
              *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
```

Acesso através do usuário ALUNO1: ok. Já o acesso através de ALUNO2 não foi bem sucedido. Isso ocorreu porque apenas criamos o sinônimo para a tabela e, embora, seja um sinônimo PUBLIC, ele não dá o direito de acesso. Vamos voltar ao usuário TSQL e dar GRANT para ALUNO2.

```
SQL> conn tsql/tsql@xe2;
Connected.
SQL>
SQL> grant select on emp to aluno2;

Grant succeeded.

SQL>
```

Vamos ao teste, novamente:

```
SQL> conn aluno2/aluno2@xe2;
Connected.
```

```
SQL>
SQL> select * from funcionarios;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- ---------- --------- ----------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM     DEPTNO
--------- ----------
                  20
      300         30
      500         30
                  20
     1400         30
                  30
                  10
                  20
                  10
        0         30
                  20
                  30
                  20
                  10

14 rows selected.
```

```
SQL>
```

Note que não foi necessário criar um sinônimo específico para o usuário ALUNO2. Ele usou o sinônimo público criado. Lembre-se também que não é o sinônimo que permite o acesso e, sim, o GRANT. Veja que este usuário pode acessar diretamente a tabela, colocando o nome do dono na frente. Isso graças à concessão criada:

```
SQL> select * from tsql.emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
                 20
     1400        30
                 30
                 10
                 20
                 10
        0        30
```

```
                    20
                    30
                    20
                    10

14 rows selected.

SQL>
```

GRANTS são criados através do usuário dono do objeto, ou por meio de um usuário com perfil de DBA, como o usuário SYSTEM. Já os sinônimos podem ser criados diretamente no usuário que deseja ter o acesso.

Informações sobre SINÔNIMOS

- ALL_SYNONYMS
- USER_SYNONYMS
- DBA_SYNONYMS

Se um sinônimo for criado para um usuário que já possui um objeto com o mesmo nome, é gerado um erro e o sinônimo não é criado. O Oracle informa que já existe um objeto criado com este nome para o usuário em questão. Isso acontece tanto quando o usuário está criando o próprio sinônimo, quanto para outro usuário que esteja criando o sinônimo para ele.

Se o sinônimo estiver sendo criado em nível público (PUBLIC), o Oracle permite a criação mesmo tendo usuários com objetos com o mesmo nome. Nesse caso, o usuário que possui um objeto com o mesmo nome do sinônimo criado, ao tentar acessar utilizando este nome, estará acessando o seu próprio objeto, e não o sinônimo. Se o usuário necessitar acessar o sinônimo, terá que indicar o nome do dono do sinônimo na frente do mesmo.

12.4 SEQUENCES

O objeto T `SEQUENCE` é utilizado como meio de incrementar valores. A Oracle disponibiliza a criação deste objeto que você pode utilizar para diversos fins, não somente para o incremento em colunas de tabelas (embora, seja sua maior utilidade), mas para qualquer finalidade em que se tenha a necessidade de gerar números sequenciais. Você pode utilizá-la para gerar numeração para recibos de pagamento, notas fiscais, etiquetas ou códigos de produtos, por exemplo. É um recurso bastante interessante, pois você não precisa controlar a sequência, o Oracle já faz isto. Você cria o objeto e ele fica armazenado no banco de dados e, quando quiser usá-lo dentro dos programas ou comandos SQL, só é preciso chamá-lo.

Quando você constrói uma `SEQUENCE`, alguns parâmetros podem ser definidos, como valor mínimo e máximo a que esta `SEQUENCE` pode chegar, se o incremento vai ser de um ou mais números por vez, e se ela poderá reiniciar ao chegar ao valor máximo. Veja a lista de parâmetros:

- **INCREMENT BY n :** especifica o intervalo entre números de sequência, onde n é um inteiro (se essa cláusula for omitida, a sequência é aumentada em incrementos de 1);
- **START WITH n :** especifica o primeiro número da sequência a ser gerado (se essa cláusula for omitida, a sequência iniciará com 1);
- **MAXVALUE n :** especifica o valor máximo que a sequência pode gerar;
- **NOMAXVALUE :** especifica o valor máximo 10`para uma sequência crescente e -1 para uma sequência decrescente (esta é a opção padrão);
- **MINVALUE n :** especifica o valor da sequência mínima;
- **NOMINVALUE :** especifica um valor mínimo igual a 1 para uma sequência crescente e -(10`) para uma sequência decrescente (esta é a opção padrão);
- **CYCLE / NOCYCLE :** especifica se a sequência continuará gerando valores depois de atingir o valor máximo ou mínimo (`NOCYCLE` é a opção padrão);

- **CACHE n / NOCACHE :** especifica quantos valores são pré-alocados pelo Oracle Server e mantidos na memória (por padrão, o Oracle Server armazena 20 valores em cache).

Segue um exemplo de criação e uso de `SEQUENCE`:

```
SQL> create sequence emp_s
     minvalue 1
     maxvalue 1000
     increment by 1
     nocycle
     noorder
     nocache
     /

Sequence created.

SQL>
```

Para incrementar a sequência:

```
SQL> select emp_s.nextval from dual;

NEXTVAL
---------
1

SQL>
```

Para mostrar o valor corrente:

```
SQL> select emp_s.currval from dual;

CURRVAL
---------
1

SQL>
```

> CURRVAL só é permitido depois que pelo menos um NEXTVAL tenha sido executado. Caso contrário, o Oracle gera um erro.

Uso no INSERT:

```
SQL> insert into emp values(emp_s.nextval,'JOCA','MECANICO',
     NULL,SYSDATE,5000,0,10);

1 row created.

SQL>

SQL> select * from emp;

    EMPNO ENAME      JOB              MGR HIREDATE         SAL
--------- ---------- --------- --------- --------- ---------
     7369 SMITH      CLERK           7902 17-DEC-80        800
     7499 ALLEN      SALESMAN        7698 20-FEB-81       1600
     7521 WARD       SALESMAN        7698 22-FEB-81       1250
     7566 JONES      MANAGER         7839 02-APR-81     3272.5
     7654 MARTIN     SALESMAN        7698 28-SEP-81       1250
     7698 BLAKE      MANAGER         7839 01-MAY-81       2850
     7782 CLARK      MANAGER         7839 09-JUN-81       2450
     7788 SCOTT      ANALYST         7566 09-DEC-82       3000
     7839 KING       PRESIDENT            17-NOV-81       5000
     7844 TURNER     VENDEDOR        7698 08-SEP-81       1500
     7876 ADAMS      CLERK           7788 12-JAN-83       1100
     7900 JAMES      CLERK           7698 03-DEC-81        950
     7902 FORD       ANALYST         7566 03-DEC-81       3000
     7934 MILLER     CLERK           7782 23-JAN-82       1300
      555 EDUARDO    ANALISTA             01-JAN-11       5000
        2 JOCA       MECANICO             21-APR-11       5000

     COMM    DEPTNO
--------- ---------
                 20
      300        30
      500        30
```

```
                 20
      1400       30
                 30
                 10
                 20
                 10
         0       30
                 20
                 30
                 20
                 10
         0       10

16 rows selected.

SQL>
```

Depois que a `SEQUENCE` gera um número, ela não passa mais pelo mesmo valor, a menos que ela reinicie. Também não há um meio de voltar o valor gerado. O que pode ser feito é adiantar a `SEQUENCE`. Com a ajuda do PL/SQL, você pode criar um script para adiantá-la. Contudo, ela precisa estar configurada para reiniciar ao chegar ao fim. Nesse caso, sua contagem será reiniciada e, desta forma, conseguiremos deixá-la na sequência anterior à que queremos. Veja os exemplos:

Exemplo 1 (com base na SEQUENCE EMP_S criada anteriormente)

Execute este `SELECT` até o valor corrente ficar maior que o valor que desejamos retornar. Para este caso, vamos deixar a sequência em 21. Nosso objetivo é voltar para sequência 5.

```
SQL> select emp_s.nextval from dual;

NEXTVAL
---------
       21

SQL>
```

Altere o `INCREMENT BY` da `SEQUENCE` com o seguinte valor: `-(NEXTVAL - 5)`, ou seja, -16.

```
SQL> alter sequence emp_s increment by -16;

Sequence altered.

SQL>
```

Execute um `NEXTVAL` para a `SEQUENCE`:

```
SQL> select emp_s.nextval from dual;

NEXTVAL
---------
5

SQL>
```

Agora, volte o `INCREMENT BY` para o valor original:

```
SQL> alter sequence emp_s increment by 1;

Sequence altered.

SQL>
```

Verifique o valor corrente da `SEQUENCE`:

```
SQL> select emp_s.currval from dual;

CURRVAL
---------
5

SQL>
```

Exemplo 2 (com base na SEQUENCE EMP_S criada anteriormente)

Execute este SELECT até o valor corrente ficar maior que o valor que desejamos retornar. Nosso objetivo é voltar para sequência 5.

```
SQL> select emp_s.nextval from dual;

NEXTVAL
---------
7

SQL>
```

Vamos alterar a SEQUENCE para CYCLE.

```
SQL> alter sequence emp_s cycle;

Sequence altered.

SQL>
```

Executamos o PL/SQL a seguir atribuindo para a variável WPOS_DESEJADA o valor inicial que desejamos para a SEQUENCE:

```
SQL> declare
       wnext_val      number := 0;
       wcurr_val      number := 0;
       wmax_value     number := 0;
       wlast_number   number := 0;
       wincrement_by number := 0;
       --
       wpos_desejada number := 5;
     begin
       --
       select max_value + wpos_desejada
             ,last_number
             ,increment_by
       into   wmax_value
             ,wlast_number
             ,wincrement_by
```

```
    from   all_sequences
    where  sequence_name = 'EMP_S';
    --
    dbms_output.put_line(wmax_value);
    dbms_output.put_line(wlast_number);
    --
    loop
      wlast_number := wlast_number + wincrement_by;
      --
      select emp_s.nextval into wnext_val from dual;
      --
      exit when wlast_number = wmax_value + 1;
      --
    end loop;
    --
    select emp_s.currval into wcurr_val from dual;
    --
    dbms_output.put_line(wcurr_val);
    --
  end;
  /

PL/SQL procedure successfully completed.

SQL>
```

Agora, vamos alterar a `SEQUENCE` para `NOCYCLE`, voltando à configuração original:

```
SQL> alter sequence emp_snocycle;

Sequence altered.

SQL>
```

Vamos ver o resultado:

```
SQL> select emp_s.currval from dual;

CURRVAL
```

```
---------
5

SQL>
```

Excluindo uma SEQUENCE:

```
SQL> drop sequence emp_s;

Sequence dropped.

SQL>
```

Informações sobre SEQUENCES

- ALL_SEQUENCES
- USER_SEQUENCES
- DBA_SEQUENCES

Cache de valores da SEQUENCE

Ao habilitar o armazenamento em cache para os valores de uma SEQUENCE, você está tornando mais ágil o acesso a estes valores. O que se deve ter em mente quando se está trabalhando com armazenamento em cache é que podem ocorrer intervalos nos valores da SEQUENCE no caso de ocorrer um *rollback*, alguma falha no sistema ou uma SEQUENCE estiver sendo usada em outra tabela.

Capítulo 13

ANEXOS

Schema SCOTT
DEPT (Departamentos)
DEPTNO (Código do Departamento) (PK)
DNAME (Nome do Departamento)
LOC (Localização do Departamento)
dept.deptno = emp.deptno
DEPT_EMP_FK
EMP (Empregados)
EMPNO (Código do Empregado) (PK)
ENAME (Nome do Empregado)
JOB (Cargo)
MGR (Código do Gerente) (FK)
HIREDATE (Data de Admissão)
SAL (Salário)
COMM (Comissão)
DEPTNO (Código do Departamento) (FK)
emp.empno = emp.mgr
EMP_MGR_FK

Schema HR

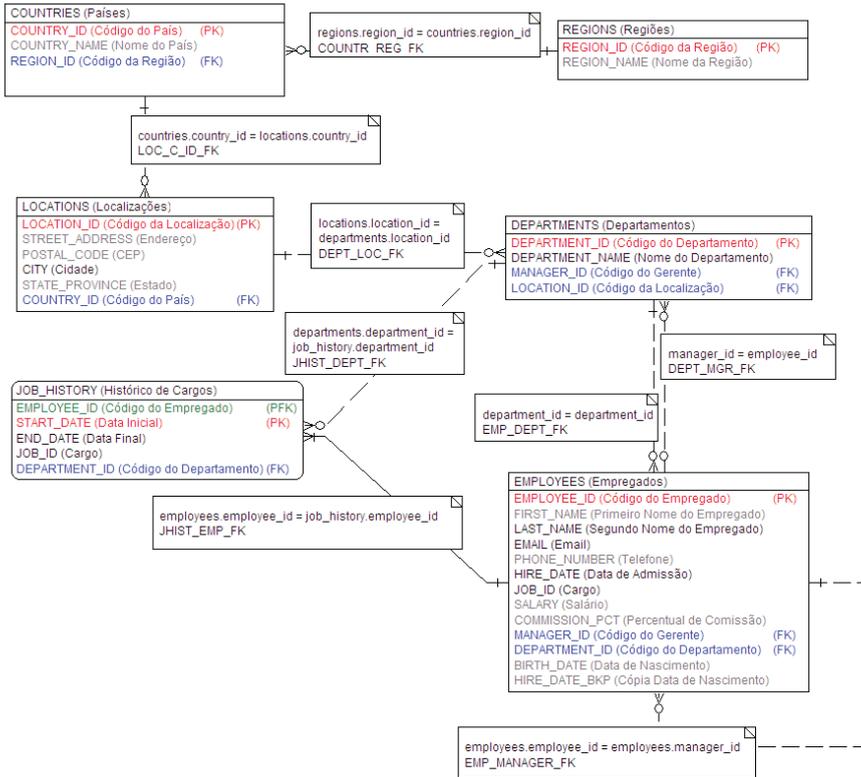



Fig. 13.2

Capítulo 14

REFERÊNCIAS BIBLIOGRÁFICAS

DATE, C. J. Introdução a Sistemas de Bancos de Dados. Rio de Janeiro: Campus, 1991.

FERNANDES, Lúcia. Oracle 9i Para Desenvolvedores Curso Completo. Rio de Janeiro: Axcel Books, 2002.

GENNICK, Jonathan, LUERS, Tom. Aprenda em 21 dias PL/SQL. 2. ed. Rio de Janeiro: Campus, 2000.

HEUSER, Carlos Alberto. Projeto de Banco de Dados. 4. ed. Porto Alegre: Sagra Luzzatto, 2001.

LIMA, Adilson da Silva. Erwin 4.0 Modelagem de Dados. 2. ed. São Paulo: Érica, 2002.

LONEY, Kevin et al. Oracle 9i : O Manual do DBA. Rio de Janeiro: Cam-

pus, 2002.

MOLINARI, Leonardo. BTO Otimização da Tecnologia de Negócio. São Paulo: Érica, 2003.

ORACLE Database Online Documentation 10g Release 1 (10.1). Disponível em: <http://www.oracle.com/pls/db10g/portal.portal_demo3?selected=1> . Acesso em: 16 mar. 2011.

Oracle Database New Features Guide 11g, Release 1 (11.1)

Oracle Database Reference 11g, Release 1 (11.1)

Oracle Database SQL Language Reference 11g, Release 1 (11.1)

Oracle Database Concepts 11g, Release 1 (11.1)

Oracle Database SQL Developer User's Guide, Release 1.2